Joana Filipa Fernandes Martins

Formula-Tree Method Tool

Department of Computer Science
Faculty of Sciences, University of Porto
28 September 2016

Joana Filipa Fernandes Martins

Formula-Tree Method Tool

Dissertation submitted to the Faculty of Sciences, University of Porto as part of the requirements for obtaining the Masters' degree in Network and Information Systems Engineering

Advisor: Professor Dr. Sabine Babette Broda
Co-advisor: Professor Dr. Sandra Maria Mendes Alves

Department of Computer Science
Faculty of Sciences, University of Porto
28 September 2016

Aos meus pais...

Acknowledgements

Ainda me recordo quando fui, a medo, falar com a professora Sabine Broda, perguntar-lhe se poderia ser ela a orientar-me a tese. Recordo-me do alívio, e da felicidade, que senti quando a professora respondeu que sim! E por isso mesmo, quero começar por agradecer ás professoras Sabine Broda e Sandra Alves por me terem dado esta oportunidade, por todo o apoio, por toda a ajuda, por todas as horas que dedicaram a mim e a esta tese. Foram incansáveis, e eu sei que tive mesmo muita sorte por vos ter como minhas orientadoras e agradeço muito tudo o que fizeram por mim. Foi uma experiência muito boa, aprendi imenso e, apesar da minha luta com a escrita e de os últimos meses terem sido mais complicados, estou muito feliz com o resultado final. Sei que vou ter saudades das nossas reuniões... Muito obrigada!

Catarina, muito obrigada pela paciência, por fazeres o esforço de perceber o que escrevi, pela tarde que perdeste na aventura do php e por toda a ajuda que me deste! Catarina, Patrícia, Danny e João, vocês estiveram sempre por perto nestes últimos anos, nos bons e nos maus momentos. Obrigada!

Xico! Foste sem dúvida o meu maior apoio, a minha maior ajuda ao longo destes anos. Não há palavras para agradecer tudo o que fizeste por mim, não há palavras para agradecer o facto de teres aturado todos os meus momentos menos bons, não há palavras para agradecer o facto de estares sempre comigo. Estes últimos meses foram os mais difíceis, não te pude dar tanta atenção quanto merecias mas, ainda assim, ajudaste-me sempre, fosse a corrigir o meu inglês, ou a apoiar-me quando as coisas não corriam como eu queria. Obrigada!

Os meus pais... Sem vocês eu não me tinha tornado na pessoa que sou, não tinha alcançado o que alcancei. Eu só posso agradecer, do fundo do meu coração, tudo o que fizeram por mim e desejar que um dia vos consiga retribuir todo o esforço que fizeram para me garantir um futuro melhor. Foram vocês quem mais sofreram com o meu mau humor quando as coisas correram menos bem, foram vocês que não me deixaram desistir quando em momentos achei que não era capaz. Obrigada por acreditarem sempre em mim. Espero que sintam tanto orgulho em mim, como eu sinto por vos ter como pais!

Abstract

This thesis describes the development of the Formula-Tree Method Tool (FTM-Tool), which assists users to address problems related to type-inhabitation in the simply typed λ-calculus, in an interactive way. To this end, the tool explores the potential of the Formula-Tree Method, which was first presented in 2000.

The FTM-Tool provides support for studying type inhabitation in the complete system of λ-calculus, also called the **SK**-calculus, as well as in three different subsystems, the **BCIW**-, the **BCI**-, and the **BCK**-calculus. Via the Curry-Howard isomorphism, types in the simply typed λ-calculus correspond to formulas in the implicational fragment of intuitionistic propositional logic, and their inhabitants correspond to proofs. Thus, the FTM-Tool can be seen as an instrument for studying provability of formulas.

Resumo

Esta tese descreve o desenvolvimento da Formula-Tree Method Tool (FTM-Tool), ferramenta que ajuda o utilizador a lidar com problemas relacionados com habitação de tipos no sistema de tipos simples do λ-calculus, de uma forma interativa. Para isso, a ferramenta explora o potencial do Formula-Tree Method, que foi apresentado pela primeira vez em 2000.

A FTM-Tool assiste o utilizador no estudo de habitação de tipos, no sistema completo do λ-calculus, também chamado **SK**-calculus, assim como em três diferentes subsystems, o **BCIW**-, o **BCI**- e o **BCK**-calculus. Via o isomorfismo de Curry-Howard, tipos no sistema simples do λ-calculus correspondem a fórmulas no fragmento implicacional da lógica proposicional intuicionista, e os seus habitantes correspondem a provas. Assim, a FTM-Tool pode ser vista como um instrumento para o estudo da provabilidade de fórmulas.

Contents

List of Figures

Chapter 1

Introduction

The objective of this dissertation is to design and implement an interactive tool, the Formula-Tree Method Tool (FTM-Tool) [2], that explores the potential of the Formula-Tree Method, in order to address problems related to type inhabitation in the simply typed λ-calculus. This problem deals with associating to a type a term (inhabitant) in the inference system of the simply typed λ-calculus. It is equivalent to the one of provability of formulas in the implicational fragment of propositional intuitionistic logic. In fact, implicational formulas and simple types are syntactically identical, and every inhabitant of a type α may be regarded as a proof of the formula α through the Curry-Howard isomorphism [25].

Given a type α, we want to solve problems such as deciding if α has an inhabitant, determine whether the number of normal inhabitants of α is finite or infinite, or compute the set of all long inhabitants of α and their η-families. These problems have been repeatedly addressed over the years [4, 6, 7, 10, 13, 14, 15, 23], both in terms of λ-calculus as well as proof theory. In [7, 10] a new formal method for exploring type inhabitation, called the Formula-Tree Method, has been presented, which proved to be effective in establishing new results as well as simplifying existing proofs of others [9, 12, 11].

The FTM-Tool takes as input a simple type, and starts by constructing the formula-tree of that type. The formula-tree of a simple type is an alternative tree-like representation, where the type is split into primitive parts, and where the formula-tree defines some kind of hierarchy between these primitive parts. The primitive parts in the formula-tree can be used to construct proof-trees, which are a compact representation of finite sets of inhabitants of the given type. In the following we will list the features

of the tool:

- Generation of formula-trees;
- Provides support for the construction of proof-trees, which are built by combining the primitive parts in the formula-tree like pieces of a puzzle, respecting the hierarchy between them;
- Provides support in the construction of proof-trees in the complete system, the **SK**-calculus, or in either one of three different subsystems, the **BCIW**-, the **BCI**-, or the **BCK**-calculus;
- From a proof-tree constructs a term-scheme from which a finite set of long normal inhabitants of the type and their η-families can be generated;
- Automatic creation of proof-trees for each one of the three subsystems, as well as minimal proof-trees within the complete system;
- Calculates if a type has an inhabitant and if the number of normal inhabitants is finite or infinite;
- Supports the user in the construction of proof-trees, such that the corresponding long inhabitants are principal inhabitants of the type;
- If the constructed proof-tree does not correspond to a principal inhabitant, the application has the option to generate the formula-tree of the type for which the generated term-scheme is a principal inhabitant;
- Computation of a context-free grammar for a type, from which all its normal inhabitants can be obtained;
- Verifies if a λ-term inhabits a type, builds the proof-tree of that λ-term, and computes the corresponding long inhabitants and their η-families.

The FTM-Tool aims to update and improve a similar application developed almost fifteen years ago [1]. The new tool updates the old one in terms of technology and improves it by adding new features. It also allows the user to work in specific subsystems of λ-calculus. In the following we list the features which are common the two applications.

Common Features:

- Generation of formula-trees;
- Provides support for the construction of proof-trees in the **SK**-calculus;
- Computation of long inhabitants and their η-families from a given proof-tree;
- Provides support for the construction of principal long normal inhabitants;
- Verification of whether a λ-term is an inhabitant.

Overview

As previously stated, the focus of this dissertation is the development of the Formula-Tree Method tool. To understand the Formula-Tree Method, one needs to be familiarized with basic notions of the λ-calculus. These notions are presented in Chapter 2 of this dissertation. The Formula-Tree Method is described in Chapter 3. In Chapter 4, we give an overview of the FTM-Tool and describe the implementation of formula-trees and proof-trees in the complete system. Furthermore, we describe the generation of minimal proof-trees, and the computation of long inhabitants and their η-families. We also explain the restrictions of each sub-system, and how these restrictions and the automatic generation of proof-trees were implemented. Chapter 5 details what is necessary for an inhabitant to be principal and explains how this feature was implemented. We explain how the tool generates a context-free grammar for a type, from which all its normal inhabitants can be obtained. Moreover we explain how the tool verifies if a λ-term inhabits a type, how it builds its proof-tree, and computes its long inhabitants and their η-family. Finally, in Chapter 6, we draw conclusions, and discuss possible future work.

Chapter 2

Background

The λ-calculus was introduced by Alonzo Church in [16] with the objective of formalizing the concept of effective computability. The origin of the λ-calculus, in the context of typed λ-calculus, dates back to the first half of the 20th century [17, 18], and has been studied since then because of its importance to mathematical logic and computer science. For a detailed reference on the λ-calculus we refer to [3].

In this chapter we will introduce the basics of the λ-calculus and the typed λ-calculus. Furthermore, we will describe three subsystems of the λ-calculus as well as related implicational logics.

2.1 The λ-Calculus

In the following we introduce the basic concepts of the λ-calculus, starting with the notion of λ-term.

Definition 2.1.1. Given an infinite enumerable set of variables $\mathcal{V}$, the set of λ-terms, denoted by Λ, is defined inductively as follows:

- $x \in \mathcal{V} \Rightarrow x \in \Lambda$;
- $x \in \mathcal{V}, M \in \Lambda \Rightarrow (\lambda x M) \in \Lambda$ (abstraction);
- $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$ (application).

We can use the following abbreviations to simplify notation,

$$\lambda x_1 \cdots \lambda x_n.M = \lambda x_1(\lambda x_2(\cdots(\lambda x_n M)\cdots))$$

and

$$MN_1 \cdots N_n = (\cdots((MN_1)N_2)\cdots N_n).$$

We will now present the notion of free variables.

Definition 2.1.2. The set of free variables of $M \in \Lambda$, denoted by $\mathsf{FV}(M)$, is defined inductively as follows:

- $\mathsf{FV}(x) = \{x\}$;
- $\mathsf{FV}(\lambda x.M) = \mathsf{FV}(M)\backslash\{x\}$;
- $\mathsf{FV}(MN) = \mathsf{FV}(M) \cup \mathsf{FV}(N)$.

A term is closed if and only if $\mathsf{FV}(M) = \emptyset$. The set of all closed λ-terms is denoted by $\Lambda^0 \subseteq \Lambda$.

We say that an occurrence of x in M is bound if and only if it occurs in a subterm of M of the form $\lambda x.P$. Occurrences that are not bound are called free occurrences. A variable can occur both free and bound in a λ-term.

Example 2.1.1. *Both occurrences of x in $\lambda x.xy$ are bound and the occurrence of y is free. In $(\lambda x.x)(\lambda y.yx)$ x occurs both bound and free.*

Intuitively, we think of a λ-term of the form $\lambda x.M$ as a function with formal parameter x. As such, the term $(\lambda x.M)N$, i.e. the application of function $\lambda x.M$ to the term N, should evaluate to the result of substituting all free occurrences of x in M by N. This type of substitution is only allowed, and in that case we say that x is free for N in M, if x does not occur free in any of M subterms of the form $(\lambda y.N)$, such that $y \in \mathsf{FV}(N)$. Note that it is always possible to change the names of bound variables in M, such that x is free for N in the resulting (equivalent) term. Terms differing only in the names of bound variables are called α-equivalent. Considering the observations above, we will from now on consider λ-terms module α-equivalence.

In the following we introduce the notion of β-reduction.

Definition 2.1.3. A term of the form $(\lambda x.M)N$ is called a β-redex and $M[N/x]$ is its β-contractum. We reduce a λ-term M in one step of β-reduction to N, and write $M \to_{1\beta} N$, if N is the result of replacing a β-redex in M by its β-contractum. The relation $\to_\beta$ is the reflexive and transitive closure of $\to_{1\beta}$, and $=_\beta$ is the reflexive, symmetric and transitive closure of $\to_{1\beta}$.

Note that the process of β-reduction can be repeated as many times as we like or until there are no more β-redexes to reduce. We say that a λ-term $M \in \Delta$ is in β-normal form (simply normal form or β-nf), if and only if it does not contain any β-redex. We say that M admits a β-nf N, if N is in β-nf and $M \to_\beta N$.

In the following we see an example of β-reduction:

Example 2.1.2. *The λ-term $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$ can be reduced as follows, where we underline each β-redex just before reducing it,*

$$\underline{(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))} \to_{1\beta} (\lambda y.y)$$

or as follows

$$(\lambda xy.y)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_{1\beta} (\lambda xy.y)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_{1\beta} \cdots$$

As we can see in the example, reducing a β-redex can create new β-redexes or remove some β-redexes. Furthermore, we can also choose to reduce the same term differently, by choosing the β-redexes in different order. Moreover, some terms can be reduced forever without reaching a β-normal form, meaning that depending on the order in which we choose the β-redexes in the reduction, we may or may not obtain a β-normal form. However, it is well known that if a term admits a β-nf then this β-nf is unique. Furthermore, terms in β-nfs can be characterized syntactically as follows.

Lemma 2.1.1. *All λ-terms in β-normal form are of the form*

$$\lambda x_1 \cdots x_n.yN_1 \cdots N_m$$

with $n, m \geq 0, x_1, \cdots, x_n, y \in \mathcal{V}$ and such that $N_1, \cdots, N_m \in \Lambda$ are in β-normal form.

The notion of η-conversion expresses the idea of extensionality in the λ-calculus, which means that two functions are considered the same if and only if they produce the same result when given the same arguments.

Definition 2.1.4 (η-conversion)**.** Let x be a variable and M a term, then

$$\lambda x.Mx \to_{1\eta} M \qquad \text{if } x \notin \mathsf{FV}(M).$$

Relations $\to_\eta$ and $=_\eta$ are defined similarly to $\to_\beta$ and $=_\beta$, following Definition 2.1.3.

2.2 Typed λ-Calculus

In the untyped λ-calculus we spoke of functions without talking about their domains and codomains. The domain and codomain of any function was the set of all λ-terms. Now we will introduce types into the λ-calculus and the notion of domain and codomain for functions. The initial motivation to define typed versions of the λ-calculus was to avoid paradoxical uses of the untyped λ-calculus [17].

In this section we describe the Curry system of simply typed λ-calculus, starting by describing the set of types $\mathbb{T}$.

Definition 2.2.1. Given a set of type variables $\mathbb{V}$, the set of simple types, denoted by $\mathbb{T}$, is inductively defined from $\mathbb{V}$ and from the connective $\to$ in the following way:

- $a \in \mathbb{V} \Rightarrow a \in \mathbb{T}$;
- $\alpha, \beta \in \mathbb{T} \Rightarrow (\alpha \to \beta) \in \mathbb{T}$.

We use $a, b, c, \cdots$ to represent type variables and greek letters $\alpha, \beta, \gamma, \cdots$ for arbitrary types. For the connective $\to$ we will consider right associativity. Thus, if $\alpha_1, \alpha_2, \cdots, \alpha_n \in \mathbb{T}$, then

$$(\alpha_1 \to (\alpha_2 \to \cdots \to (\alpha_{n-1} \to \alpha_n) \cdots))$$

can be represented as

$$\alpha_1 \to \alpha_2 \to \cdots \to \alpha_n.$$

Every type α can be uniquely written as

$$\alpha = \alpha_1 \to \cdots \to \alpha_n \to a$$

with $n \geq 0$. The type-variable a is called the tail of α and denoted as $\mathsf{tail}(\alpha)$. Furthermore, if $n \geq 1$, then $\alpha_1, \cdots, \alpha_n$ are called the arguments.

We will now define the notion of subpremise.

Definition 2.2.2. Occurrences of a type α satisfying the following conditions are called negative (resp. positive) subpremises of α as follows:

- α is a positive subpremise of α;
- if $\alpha = \alpha_1 \to \cdots \to \alpha_n \to a$, then every positive (resp. negative) subpremise of any of α_j, $1 \leq j \leq n$, is a negative (resp. positive) subpremise of α.

Note that each subpremise is a particular occurrence of a subtype and that not all occurrences of subtypes are subpremises.

Example 2.2.1. *The subpremises of the type*

$$((a \to b) \to a \to b) \to (a \to b) \to a \to b$$

are all the underlined occurrences of subtypes in

$$\underline{(\underline{\underline{(a \to b)} \to a \to b)} \to \underline{(a \to b)} \to a \to b}.$$

Note that the number of lines below a subtype is odd if it corresponds to a positive subpremise, and even if it corresponds to a negative subpremise.

We now describe the type assignment system for simple types.

Definition 2.2.3. In a declaration of the form $M : \alpha$, M is called the subject and α is called the predicate. A finite, possibly empty, set of declarations with distinct variables as subjects is called a context.

Definition 2.2.4. In the Curry type system, we say that M admits type α given the context Γ and write

$$\Gamma \vdash M : \alpha$$

if this expression can be obtained by applying the following derivation rules a finite number of times. If $\vdash M : \alpha$, then M is called an inhabitant of α.

$$\frac{}{\Gamma, x : \alpha \vdash x : \alpha} \qquad (\to \text{ Axiom})$$

$$\frac{\Gamma, x : \alpha_1 \vdash M : \alpha_2}{\Gamma \vdash \lambda x.M : \alpha_1 \to \alpha_2} \qquad (\to \text{Intro})$$

$$\frac{\Gamma \vdash M : \alpha_1 \to \alpha_2 \qquad \Gamma \vdash N : \alpha_1}{\Gamma \vdash MN : \alpha_2} \qquad (\to \text{Elim})$$

In the previous definition, $\Gamma, x : \alpha$ represents the set $\Gamma \cup \{x : \alpha\}$. One important property of these typing rules is that there is precisely one rule for each kind of λ-term. Thus, when we construct typing derivations in a bottom-up fashion, there is always a unique choice of which rule to apply next. The only real choice we have is about which types to assign to subterms, when applying the $(\to Elim)$-rule.

We will now see an example of a valid typing derivation obtained in the Curry simple type system:

Example 2.2.2.

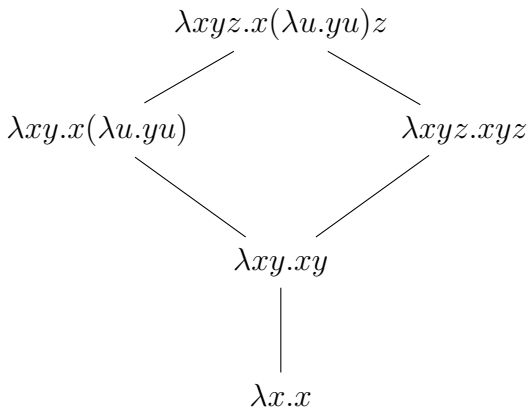
$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : b \to b, y : a \vdash x : b \to b}\,(\text{Axiom})}{x : b \to b \vdash \lambda y.x : a \to b \to b}\,(\to \text{Intro})}{\vdash \lambda xy.x : (b \to b) \to a \to b \to b}\,(\to \text{Intro}) \qquad \dfrac{\dfrac{}{x : b \vdash x : b}\,(\text{Axiom})}{\vdash \lambda x.x : b \to b}\,(\to \text{Intro})}{\vdash (\lambda xy.x)(\lambda x.x) : a \to b \to b}\,(\to \text{Elim})$$

Definition 2.2.5. A term M is called typable if there exists a context Γ and a type α such that $\Gamma \vdash M : \alpha$.

Not all λ-terms are typable. For example, $\lambda x.xx$ does not admit a type. But, is well known that all typable λ-terms admit a β-normal form. As a consequence of this property, when we are looking for inhabitants of a type, we can focus on normal forms. In fact, this search can be further restricted to a particular type of inhabitants, the long inhabitants.

Definition 2.2.6. A β-normal inhabitant of a type α is called a long normal inhabitant of α if and only if every variable-occurrence z in M is followed by the longest sequence of arguments allowed by its type, i.e. if and only if each component of the form $(zP_1 \cdots P_n), (n \geq 0)$ that is not in a function position has atomic type, which means that it has no arguments. The finite set of all terms obtained by η-reducing a λ-term M is called the η-family of M and denoted by $\{M\}_\eta$. This set is necessarily finite because there is a limited number of times that a λ-term can be η-reduced. As we can see in the next example, the term decreases in size in each reduction step.

Example 2.2.3. *The following diagram represents the η-family of the long normal inhabitant $\lambda xyz.x(\lambda u.yu)z$ of type $((a \to b) \to a \to b) \to (a \to b) \to a \to b$.*



Ben-Yelles [4, 23] showed that every normal inhabitant of a type α can be η-expanded to one unique long normal inhabitant of α. Consequently, the set of normal inhabitants of a type consists of its long normal inhabitants and their finite η-families. The number of long inhabitants can be either zero, finite or infinite.

In the following example we will illustrate the process of searching for long normal inhabitants. Type α in this example will be our running example in this dissertation.

Example 2.2.4. *Consider the type*

$$\alpha = ((a \to b) \to a \to b) \to (a \to b) \to a \to b.$$

The acronym **LNS** *corresponds to long normal search steps, for which we do not have the option to make a choice. The acronym* **CH** *corresponds to the choices we made during the search.*

LNS 0: *Every long normal inhabitant of α has to be of the form $M = \lambda xyz.M_1$, where x, y and z have respectively types $(a \to b) \to a \to b$, $a \to b$ and a. This means that we are searching for a long term M_1 such that $\Gamma \vdash M_1 : b$, where*

$$\Gamma = \{x : (a \to b) \to a \to b, y : a \to b, z : a\}.$$

> **CH a:** *Since the type of x has tail b, one possibility is to take $M_1 = xM_2M_3$, where $\Gamma \vdash M_2 : a \to b$ and $\Gamma \vdash M_3 : a$.*
>
> **LNS 1:** *To continue the search, since the type of M_2 is not atomic, M_2 has to be of the form $\lambda u_1.M_4$, where $\Gamma, u_1 : a \vdash M_4 : b$. Now, we have two types in the context with tail-variable a.*

CH 1.a: *We will repeat the process and, since the type of x has tail b, again one possibility is to take $M_4 = xM_5M_6$, where $\Gamma, u_1 : a \vdash M_5 : a \to b$ and $\Gamma, u_1 : a \vdash M_6 : a$.*

LNS 1.a.1: *Furthermore, since the type of M_5 is not atomic, M_5 has to be of the form $\lambda u_2.M_7$, where $\Gamma, u_1 : a, u_2 : a \vdash M_7 : b$. Now, we have three types in the context with tail-variable a.*

Note, that at this point we have introduced two declarations $u_1 : a$ and $u_2 : a$ in the context, both with predicate a, corresponding precisely to the same negative subpremise of α. This means that using u_1 or u_2 in the remaining search will lead to the exact same steps, and is basically the same, but for the name of the used variable.

CH 1.a.1.a: *Since the type of y has tail b, one possibility is to take $M_7 = yM_8$, where $\Gamma, u_1 : a, u_2 : a \vdash M_8 : a$.*

LNS 1.a.1.a.1: *M_8 is already of atomic type.*

CH 1.a.1.a.2.a: *At this point we have three declarations in Γ whose predicate (type) has tail-variable a, to be specific z, u_1 and u_2. We can take $M_8 = z$.*

CH 1.a.1.a.2.b: *But we can also take $M_8 = u_1$.*

CH 1.a.1.a.2.c: *Or $M_8 = u_2$.*

LNS 1.a.2: *M_6 is already of atomic type.*

CH 1.a.2.a: *Now we have two declarations in Γ whose type has tail-variable a, to be specific z and u_1. We can take $M_6 = z$.*

CH 1.a.2.b: *But we can also take $M_6 = u_1$.*

LNS 2: *M_3 is already of atomic type.*

CH 2.a: *Since z is the only variable in Γ whose type has tail-variable a, one has $M_3 = z$.*

This kind of repetitive choice, as well as the fact that M has to be of the form $\lambda xyz.M_1$, M_2 has to be of the form $\lambda u_1.M_4$, M_5 has to be of the form $\lambda u_2.M_7$, etc., i.e. steps corresponding to **LNS** *step in the proof-search above, will not be present*

in the formula-tree search method. The formula-tree search method focuses on the meaningful decisions, i.e. steps corresponding to **CH** *steps, made during the search. We will see this in detail in the next chapter.*

In the following we introduce the notion of principal type, which is the most general type of a λ-term. In fact, the principal type represents every type that can be assigned to a λ-term.

Example 2.2.5. *Consider the types $\alpha_1 = (a \to b) \to a \to b$, and $\alpha_2 = a \to a$.*

We can assign both types α_1 and α_2 to the λ-term $\lambda x.x$. Type α_1 can be obtained as an instance of α_2, which means that it is not a principal type. In fact, the most general type of $\lambda x.x$, i.e. the λ-term's principal type, is α_2. The type α_1 is the principal type of the λ-term $\lambda xy.xy$.

2.3 Subsystems and Implicational Logics

There exists a close relationship between the simply typed λ-calculus and the implicational fragment of intuitionistic propositional logic.

In the following we describe three subsystems the of λ-calculus, which impose syntactical restrictions on the λ-terms.

In the **BCIW**-calculus every term M is such that, for every subterm $\lambda x.N$ of M one has $x \in \mathsf{FV}(N)$. This fragment is also called the $\lambda_{\mathcal{I}}$-calculus.

Definition 2.3.1. Given an infinite enumerable set of variables $\mathcal{V}$, the set of **BCIW**-terms, Λ_{BCIW}, is defined by the following rules.

- $\mathcal{V} \subseteq \Lambda_{\mathcal{BCIW}}$;
- $M \in \Lambda_{\mathcal{BCIW}}, x \in \mathsf{FV}(M) \Rightarrow \lambda x.M \in \Lambda_{\mathcal{BCIW}}$;
- $M, N \in \Lambda_{\mathcal{BCIW}} \Rightarrow MN \in \Lambda_{\mathcal{BCIW}}$.

The set of closed **BCIW**-terms is denoted by $\Lambda^0_{\mathcal{BCIW}}$.

In the **BCK**-calculus every variable occurs free at most once for every subterm of a term M, and for each subterm of the form $\lambda x.N$ of M, one has at most one free occurrence of x in N. This fragment is also called the affine λ-calculus.

Definition 2.3.2. Given an infinite enumerable set of variables $\mathcal{V}$, the set of **BCK**-terms, is defined by the following rules.

- $\mathcal{V} \subseteq \Lambda_{\mathcal{BCK}}$;
- $M \in \Lambda_{\mathcal{BCK}} \Rightarrow \lambda x.M \in \Lambda_{\mathcal{BCK}}$;
- $M, N \in \Lambda_{\mathcal{BCK}}, \mathsf{FV}(M) \bigcap \mathsf{FV}(N) = \emptyset \Rightarrow MN \in \Lambda_{\mathcal{BCK}}$.

The set of closed **BCK**-terms is denoted by $\Lambda^0_{\mathcal{BCK}}$

In the **BCI**-calculus for each subterm of the form $\lambda x.N$ of M, one has exactly one free occurrence of x in N. This fragment is also called the linear λ-calculus.

Definition 2.3.3. Given an infinite enumerable set of variables $\mathcal{V}$, the set of **BCI**-terms, is defined by the following rules.

- $\mathcal{V} \subseteq \Lambda_{\mathcal{BCI}}$;
- $M \in \Lambda_{\mathcal{BCI}}, x \in \mathsf{FV}(M) \Rightarrow \lambda x.M \in \Lambda_{\mathcal{BCI}}$;
- $M, N \in \Lambda_{\mathcal{BCI}}, \mathsf{FV}(M) \bigcap \mathsf{FV}(N) = \emptyset \Rightarrow MN \in \Lambda_{\mathcal{BCI}}$.

The set of closed **BCI**-terms is denoted by $\Lambda^0_{\mathcal{BCI}}$

In the remaining we will frequently refer to the entire system as the **SK**-calculus.

Next we describe the implicational fragment of intuitionistic logic, to which we will also refer as **SK**-logic, as well as three subsystems.

Definition 2.3.4. Given a set of propositional variables $\mathbb{V}$, the set of propositional implicational formulas $\mathbb{F}$ is inductively defined as follows.

- $\mathbb{V} \subseteq \mathbb{F}$;
- $\alpha, \beta \in \mathbb{F} \Rightarrow (\alpha \to \beta) \in \mathbb{F}$.

Note that $\mathbb{F}$ and the set of simple types $\mathbb{T}$ are syntactically identical. Consequently, every implicational formula can be seen as a simple type and vice-versa.

In the following we define the set of theorems of the implicational fragment of propositional intuitionistic logic, which we will also call **SK**-logic, as well as its subsystems **BCIW**-, **BCK**- and **BCI**-logic.

Each of these subsystems is specified by a set of axiom schemes and the inference rule Modus Ponens. The set of axiom schemes relevant for the systems considered in this dissertation are the following:

- (**I**) $\alpha \to \alpha$
- (**K**) $\alpha \to \beta \to \alpha$
- (**B**) $(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$
- (**C**) $(\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma$
- (**W**) $(\alpha \to \alpha \to \beta) \to \alpha \to \beta$
- (**S**) $(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$

SK-logic is based on axiom schemes **(S)** and **(K)** and defined as follows.

Definition 2.3.5. The set of theorems of the **SK**-logic is inductively defined as follows.

- Every formula in $\mathbb{F}$ of the form **(S)** and **(K)** is a theorem of **SK**-logic;
- If $\alpha \to \beta$ and α are theorems, then β is a theorem (Modus Ponens).

Likewise the **BCIW**-logic is defined based on axiom schemes **(B)**, **(C)**, **(I)**, and **(W)**; the **BCK**-logic is based on axiom schemes **(B)**, **(C)** and **(K)**; and the **BCI**-logic is based on axiom schemes **(B)**, **(C)** and **(I)**.

The following result stabilishes the relationship between the λ-calculus and the implicational intuitionistic logic.

Proposition 2.3.1. *A simple type $\alpha \in \mathbb{T}$ is a theorem of the* **SK***-logic (respectively* **BCIW***-,* **BCK***- and* **BCI***-logic) if and only if there exists a closed term $M \in \Lambda^0$ (respectively Λ^0_{BCIW} and Λ^0_{BCK}, Λ^0_{BCI}) such that $\vdash M : \alpha$.*

Chapter 3

The Formula-Tree Method

In the Formula-Tree Method types are represented by formula-trees. The formula-tree of a type α, denoted by $\mathsf{tree}(\alpha)$, is obtained from a tree representation of α, by splitting it into primitive parts. These primitive parts correspond more or less to negative subpremises of α, that would be part of the context during the proof-search for long inhabitants. The formula-tree defines some kind of hierarchy between these primitive parts. We can look at them as if they were parts of a puzzle, which can be used to create new trees, called proof-trees. Each proof-tree of α will correspond to a term-scheme, from which a set of long inhabitants and their η-families can be obtained.

In this chapter, it will be explained how one can construct a type's formula-tree, proof-trees, long inhabitants, and their η-families.

3.1 The Formula-Tree of a Type

We start by seeing an example of how to obtain the formula-tree of a type.

Example 3.1.1. *Consider type* $\alpha = ((a \to b) \to a \to b) \to (a \to b) \to a \to b$*, and its tree representation:*

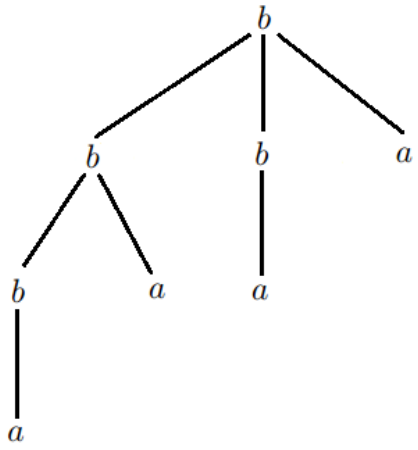


Figure 3.1: Tree Representation of Type α.

From α, we obtain the following formula-tree:

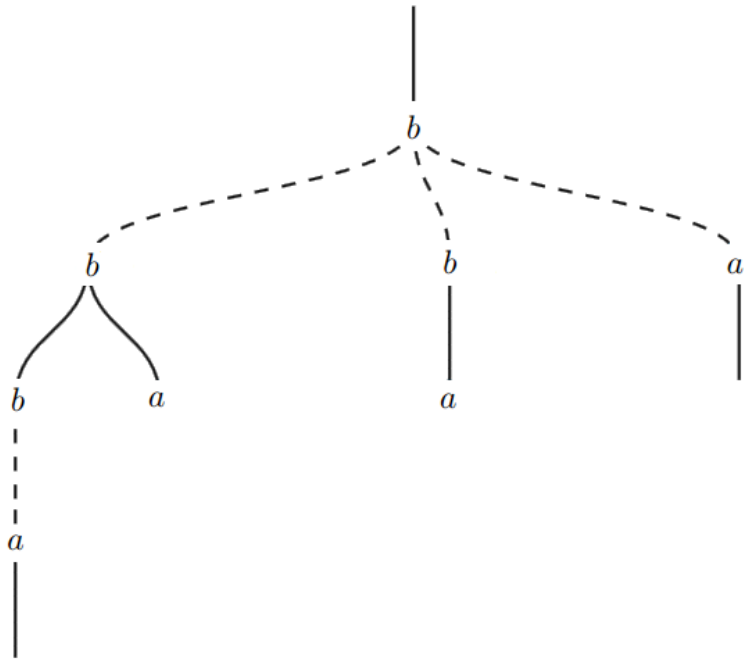


Figure 3.2: Formula-Tree of Type α.

To obtain the formula-tree of α, it is necessary to make some changes in the tree representation of the type. The first thing to do is to change the shape of lines which are at an odd level, from continuous to dashed. Then, it is necessary to add a vertical line above the root node, and below every node that has no descendants, and such that the line that connects it to its parent node is dashed. Note that it is possible to distinguish the different primitive parts in the formula-tree through the lines. Lines between nodes of the same primitive part are continuous, lines between primitive parts

are dashed. In the following we will see how the primitive parts are represented.

The primitive parts of this formula-tree are:

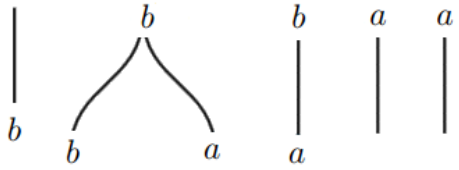
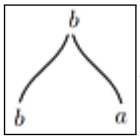


Figure 3.3: Primitive Parts of the Formula-Tree

And the hierarchy defined over them by the formula-tree establishes that there are three primitive parts descending from the tail-variable of  *and one primitive part,*  *, descending from the first (leftmost) tail-variable in*  *.*

We now provide a formal definition of an algorithm, from [10], for the construction of a formula-tree, as exemplified above.

The formula-tree of a type α, denoted by $\mathsf{tree}(\alpha)$, is an alternative tree-like representation, whose nodes are labelled by primitive parts of either one of the following forms:

(p1)

Figure 3.4: Root Node

(p2)

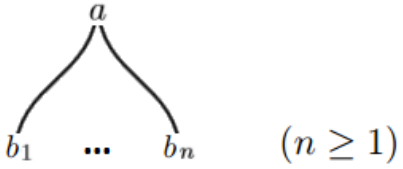


Figure 3.5: Internal Node

(p3)

Figure 3.6: Leaf Node

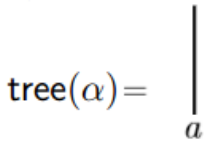
The primitive parts are named based on the role that they assume in the proof-trees. A proof-tree always begins with a root node, and every path ends with a leaf node; all other nodes are internal nodes. In these primitive parts a is called the head, and $b, b_1, \ldots, b_n$ are called tail-variables.

We refer to formula-trees as a tree-like representation, because they are not real trees in the usual sense. In a regular tree every node, which is not the root node of the tree, descends from another node. In a formula-tree all nodes but the root node descend from a specific tail-variable of a primitive part, which labels another node in the tree.

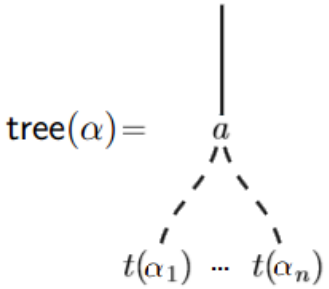
In the following, we consider a type α. Remember that α can be uniquely written in the form $\alpha = \alpha_1 \to \cdots \to \alpha_n \to a$ where a is an atom, $\alpha_1, \ldots, \alpha_n$ are types, and $n \geq 0$.

Then $\mathsf{tree}(\alpha)$ is computed as follows:

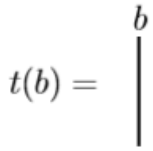
If $n = 0$, i.e. $\alpha \equiv a$, then



If $n \geq 1, then$

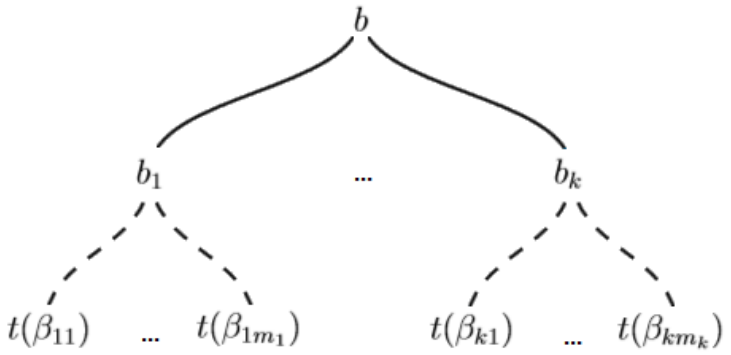


where $t(\alpha_1)$ is defined by the following.

$t(b) =$ 

For $k \geq 1$ and $m_1, \cdots, m_k \geq 0$ take

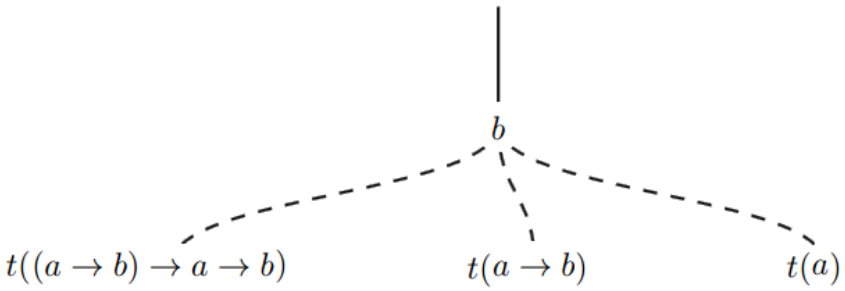
$$t((\beta_{11} \to \cdots \to \beta_{1m_1} \to b_1) \to \cdots \to (\beta_{k1} \to \beta_{km_k} \to b_k) \to b) =$$



Example 3.1.2. *Consider α from the previous example.*

The algorithm follows the following steps:

Step 1:



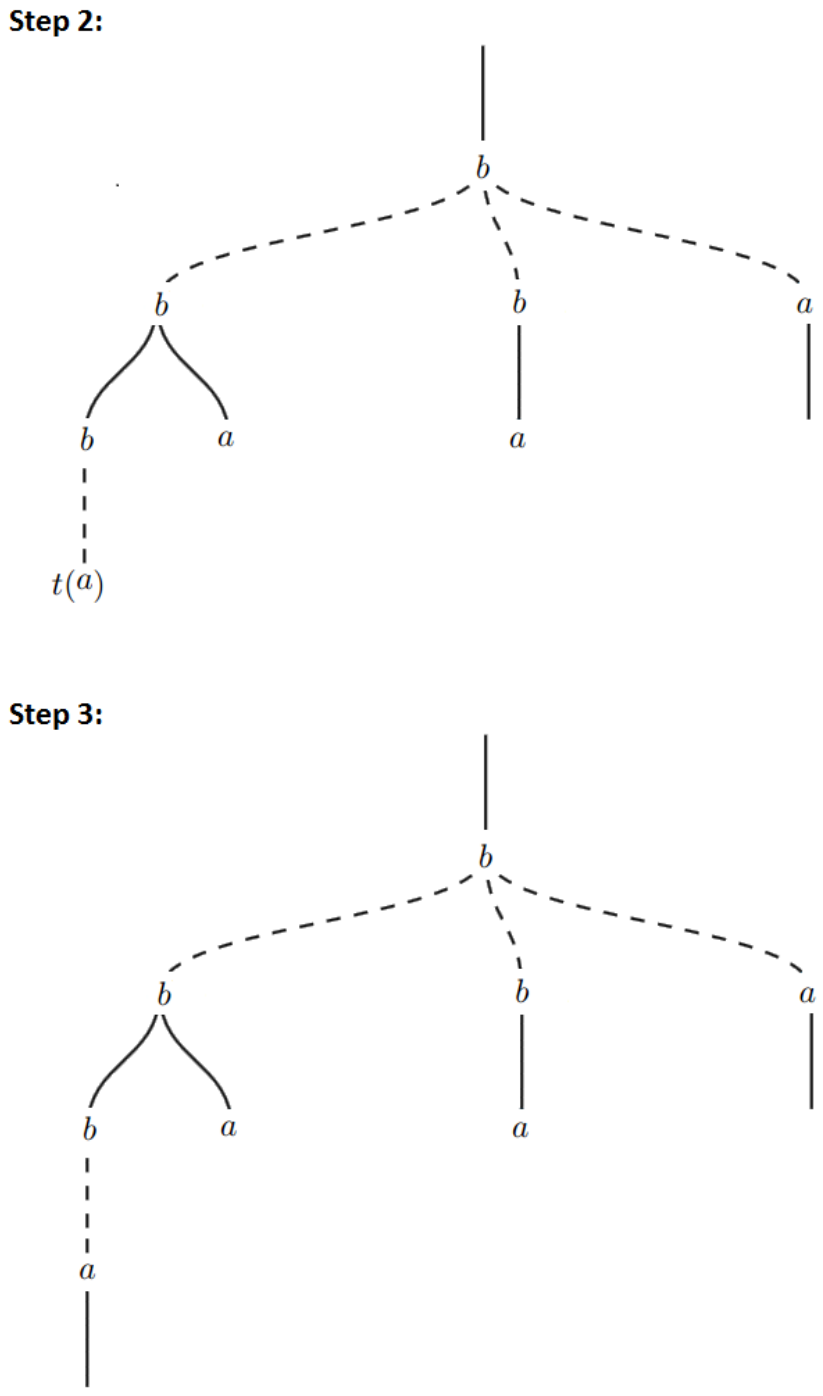


Figure 3.7: Formula-Tree Construction

Note that every primitive part together with the primitive parts descending from it, corresponds to a negative subpremise of α, that can be part of the context during the proof-search for long inhabitants.

3.2 Proof-Trees

In this section we illustrate the process of constructing proof-trees by an example.

Example 3.2.1. *Consider again α, whose formula-tree is presented in Example 3.1. There are five principal parts in* $\mathsf{tree}(\alpha)$. *We will name them, except for the one at the root node, since occurrences of primitive parts in a proof-tree will correspond to variable occurrences in the long inhabitants represented by it. For instance, every occurrence of primitive part x in a proof-tree* PT *will correspond to an occurrence of a variable x, of type $(a \to b) \to a \to b$, in every inhabitant M_{PT} obtained from* PT.

Proof-trees for α are built using the primitive parts, starting with the root node at the top. A proof-tree is closed when all of its branches end with a leaf node; i.e. primitive part z or u.

In Figure 3.8 we can see the names of the primitive parts:

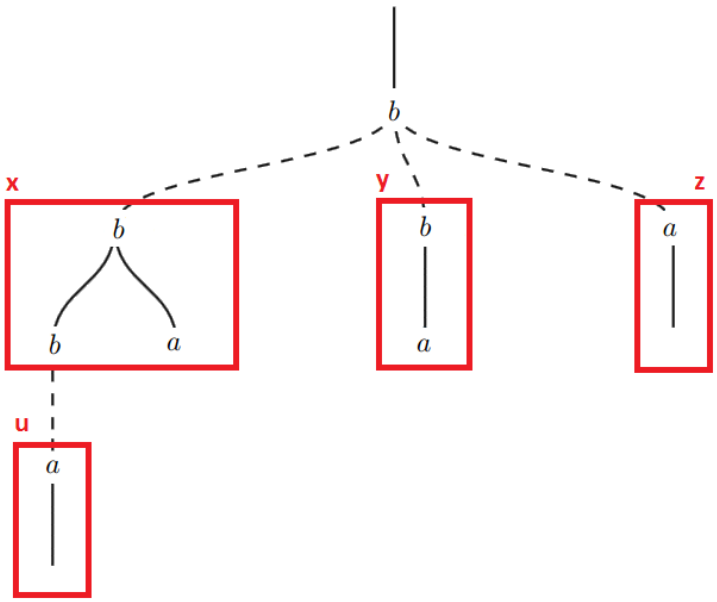
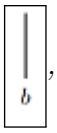


Figure 3.8: Primitive Parts Names

We can see in the previous image that the primitive part u descends from the leftmost tail-variable, b, of primitive part x. This means that u can only be used, i.e. is available, in branches which descend from the leftmost tail-variable of some occurrence of x, in the proof-tree. In particular, in the very beginning of the proof-tree construction the available parts are x, y and z.

In the beginning, i.e. starting with root node 

, we have to add a primitive part with head-variable b. This can be either x or y. Then, if the chosen primitive part is x, it is necessary to construct two complete sub-trees. One descending from the leftmost tail-variable b of primitive part x (which means that a primitive part with head-variable b has to be used next), and one descending from the rightmost tail-variable a of primitive part x (which means that a primitive part with head-variable a has to be used next).

In the left sub-tree primitive part u will be available, but it will not be available in the right sub-tree (at least for now).

After adding a few parts we can obtain, for example, the following proof-tree:

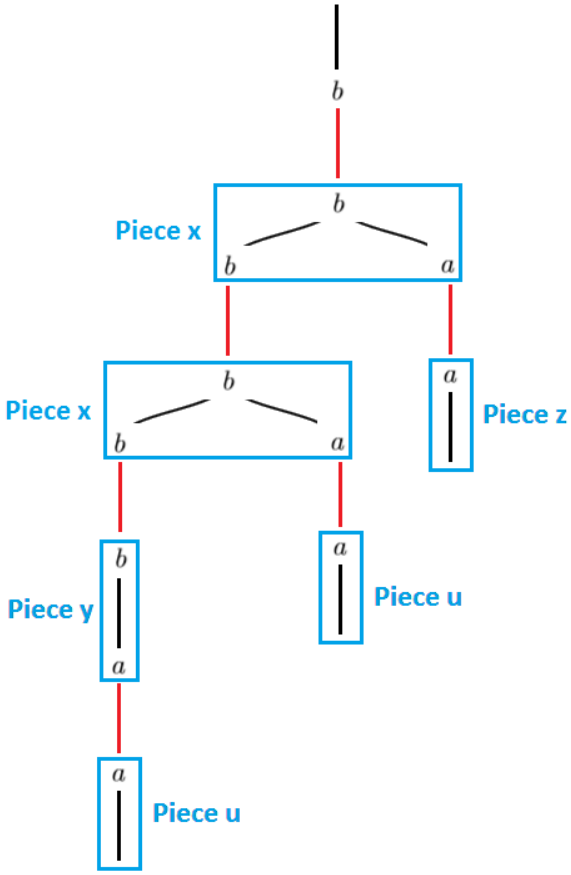


Figure 3.9: Building a Proof-Tree

During the construction of the proof-tree, x was chosen because of its head of type b, while in the proof-search of Example 2.2.4, x is chosen because it has tail b, cf. steps **CH a** *and* **CH 1.a**.

CH a: *Since the type of x has tail b, one possibility is to take $M_1 = xM_2M_3$, where $\Gamma \vdash M_2 : a \to b$ and $\Gamma \vdash M_3 : a$.*

Figure 3.10: CH a

CH 1.a: *We will repeat the process and, since the type of x has tail b, again one possibility is to take $M_4 = xM_5M_6$, where $\Gamma, u_1 : a \vdash M_5 : a \to b$ and $\Gamma, u_1 : a \vdash M_6 : a$.*

Figure 3.11: CH 1.a

In the search of a long inhabitant two simplification steps follow necessarily (which are not present in the proof-tree construction). These are **LNS 1** *and* **LNS 2**, *and* **LNS 1.a.1** *and* **LNS 1.a.2**, *respectively.*

LNS 1: *To continue the search, since the type of M_2 is not atomic, M_2 has to be of the form $\lambda u_1.M_4$, where $\Gamma, u_1 : a \vdash M_4 : b$. Now, we have two types in the context with tail-variable a.*

Figure 3.12: LNS 1

LNS 2: *M_3 is already of atomic type.*

Figure 3.13: LNS 2

LNS 1.a.1: *Furthermore, since the type of M_5 is not atomic, M_5 has to be of the form $\lambda u_2.M_7$, where $\Gamma, u_1 : a, u_2 : a \vdash M_7 : b$. Now, we have three types in the context with tail-variable a.*

Figure 3.14: LNS 1.a.1

LNS 1.a.2: *M_6 is already of atomic type.*

Figure 3.15: LNS 1.a.2

In the subsequent steps one will search for terms of type b and a, which will appear in the first and in the second argument of x. For this, one can chose between all variables available in the respective context, which is either $\Gamma = \{x : (a \to b) \to a \to b, y : a \to b, z : a\}$ or $\Gamma \cup \{u_1 : a\}$. Similarly, during the proof-tree construction in the next step two primitive parts have to be chosen, whose heads are respectively b and a, since these are the tail-variables of primitive part x. In the first case, i.e. b, one can choose between primitive parts x or y, while in the second case, i.e. for a, One has to choose z, or can choose between z or u, respectively.

In our example of a proof-tree construction y was chosen because of its head of type b, while y is chosen in the proof-search of Example 2.2.4 because it has tail b, c.f step

CH 1.a.1.a.

CH 1.a.1.a: *Since the type of y has tail b, one possibility is to take $M_7 = yM_8$, where $\Gamma, u_1 : a, u_2 : a \vdash M_8 : a$.*

Figure 3.16: CH 1.a.1.a

In the search of a long inhabitant one simplification step follows necessarily. This is **LNS 1.a.1.a.1**.

LNS 1.a.1.a.1: *M_8 is already of atomic type.*

Figure 3.17: LNS 1.a.1.a.1

Again this step is not present in the proof-tree construction.

In the subsequent steps one will search for terms of type a, which will appear in the argument of y. For this, one can chose between all variables available in the respective context, which is $\Gamma \cup \{u_1 : a, u_2 : a\}$. Similarly, during the proof-tree construction in the next step one primitive part has to be chosen, whose head is respectively a, since this is the tail-variable of primitive part y. One has to choose between z and u.

The representation above of the proof-tree is just an example so we can better understand what is happening during its construction. In the real proof-tree representation we do not have edges between parts. Each tail-variable is overlapped with the head-variable of the primitive part to which it is connected. Thus, the actual representation of the proof-tree is as follows:

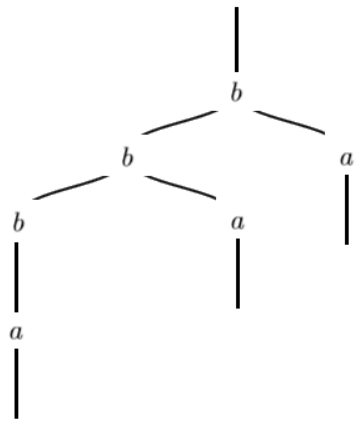


Figure 3.18: Building a Proof-Tree - Final Proof-Tree

3.3 Long Inhabitants and Their η-Families

Since the construction of a proof-tree PT mimics the construction of a (finite) set of long inhabitants $\mathsf{Inhab}(\mathsf{PT})$, it is possible to recover $\mathsf{Inhab}(\mathsf{PT})$ from PT and from tree(α). The set $\mathsf{Inhab}(\mathsf{PT})$ correspondent to PT will be represented by one unique object, called the term-scheme of PT, from which a finite set of long inhabitants of α and their η-families can be obtained. In fact, during the construction of a PT it is possible to simultaneously construct its respective term-scheme. In the following we will illustrate this process.

Example 3.3.1. *Consider α from Example 2.2.4, and the proof-tree PT from Example 3.2.1.*

Following the **LNS 0** *step, we saw that every long inhabitant of PT has to be of the form $M = \lambda xyz.M_1$. This means that we are looking for a long term M_1, such that $\Gamma \vdash M_1 : b$, where*

$$\Gamma = \{x : (a \to b) \to a \to b, y : a \to b, z : a\}.$$

In Example 3.2.1, the chosen part was x, thus $M_1 = xM_2M_3$. Applying the substitution, we can see that every long inhabitant of PT is of the form $M = \lambda xyz.xM_2M_3$. Repeating the same process as many times as needed, applying the substitutions correspondent to Example 3.2.1, the obtained term-scheme for PT is:

$$\lambda xyz.x(\lambda u.x(\lambda u.yu)u)z$$

From a proof-tree's term-scheme it is possible to obtain the correspondent long inhabitants. We can do this by, for each variable v, indexing all abstractions λv differently, and renaming occurrences of v in the scope of these abstractions in all possible ways, considering all possible alternatives in a given context. Afterwards, we can calculate their η-families by η-reducing the long inhabitants.

Example 3.3.2. *Recall Examples 2.2.4, and 3.3.1.*

From $\lambda xyz.x(\lambda u.x(\lambda u.yu)u)z$, after renaming, we obtain the following long inhabitants:

$$\lambda xyz.x(\lambda u_1.x(\lambda u_2.yu_1)u_1)z$$

$$\lambda xyz.x(\lambda u_1.x(\lambda u_2.yu_2)u_1)z$$

By η-reducing them, we calculate their η-families:

$$\eta\text{-}family(\lambda xyz.x(\lambda u_1.x(\lambda u_2.yu_1)u_1)z) = \{\lambda xyz.x(\lambda u_1.x(\lambda u_2.yu_1)u_1)z,$$
$$\lambda xy.x(\lambda u_1.x(\lambda u_2.yu_1)u_1)\}$$
$$\eta\text{-}family(\lambda xyz.x(\lambda u_1.x(\lambda u_2.yu_2)u_1)z) = \{\lambda xyz.x(\lambda u_1.x(\lambda u_2.yu_2)u_1)z,$$
$$\lambda x.xx, \lambda xy.x(\lambda u_1.x(\lambda u_2.yu_2)u_1),$$
$$\lambda xy.x(\lambda u_1.xyu_1), \lambda xy.x(x(\lambda u_2.yu_2)),$$
$$\lambda xy.xxy, \lambda xyz.x(\lambda u_1.xyu_1)z,$$
$$\lambda xyz.x(x(\lambda u_2.yu_2))z, \lambda xyz.xxyz\}$$

Now that we have described the Formula-Tree Method, we will present the tool that implements this method, the Formula-Tree Method Tool.

Chapter 4

The Formula-Tree Method Tool

In this chapter we start by giving an overview of the FTM tool. We present the technologies used in the development of the tool, and the two objects that keep the external information given to the tool. We also describe the main features of the FTM-tool, namely the construction of the formula-tree of the given type, the construction of proof-trees, and the construction and automatic generation of minimal proof-trees in the **SK**-calculus. Furthermore, we describe the construction and automatic generation of proof-trees in the subsystems presented in Section 2.3, namely the **BCIW**-, **BCI**- and **BCK**-subsystems.

4.1 Overview

In Figure 4.1 we show the flowchart of the FTM-Tool, which provides an overview of the tool's features. In the first page of the tool there are three options.

1. Build the formula-tree of the given type;

2. Obtain a context-free grammar for the type;

3. Choose a λ-term and verify if it is an inhabitant of the type.

In the first option, the formula-tree of the type is presented, and then it is possible to choose between: construct proof-trees in the **SK**-calculus; in either one of the three subsystems; or construct principal proof-trees. In the construction of proof-trees in

the **SK**-calculus there are two options available. The first one allows the construction of minimal proof-trees, while the second one allows the automatic generation of minimal proof-trees for the type. In the construction of proof-trees in each one of the subsystems, there is also an option to automatically generate proof-trees in the subsystem. As we saw in Chapter 3, for every proof-tree there is an associated term-scheme, from which the long inhabitants and their η-families are obtained. We can see in the flowchart that, whatever the chosen option is, at the end of each proof-tree construction (or automatic generation), it is possible to obtain its long inhabitants and their η-families.

In the second option, the tool obtains a context-free grammar for the given type. The language generated by this grammar is the set of all term-schemes corresponding to the type.

In the third option, given a λ-term as input, the tool verifies if the given term is an inhabitant of the type. In that case, the tool has the options to automatically generate its proof-tree, or to obtain its long inhabitants and their η-families.

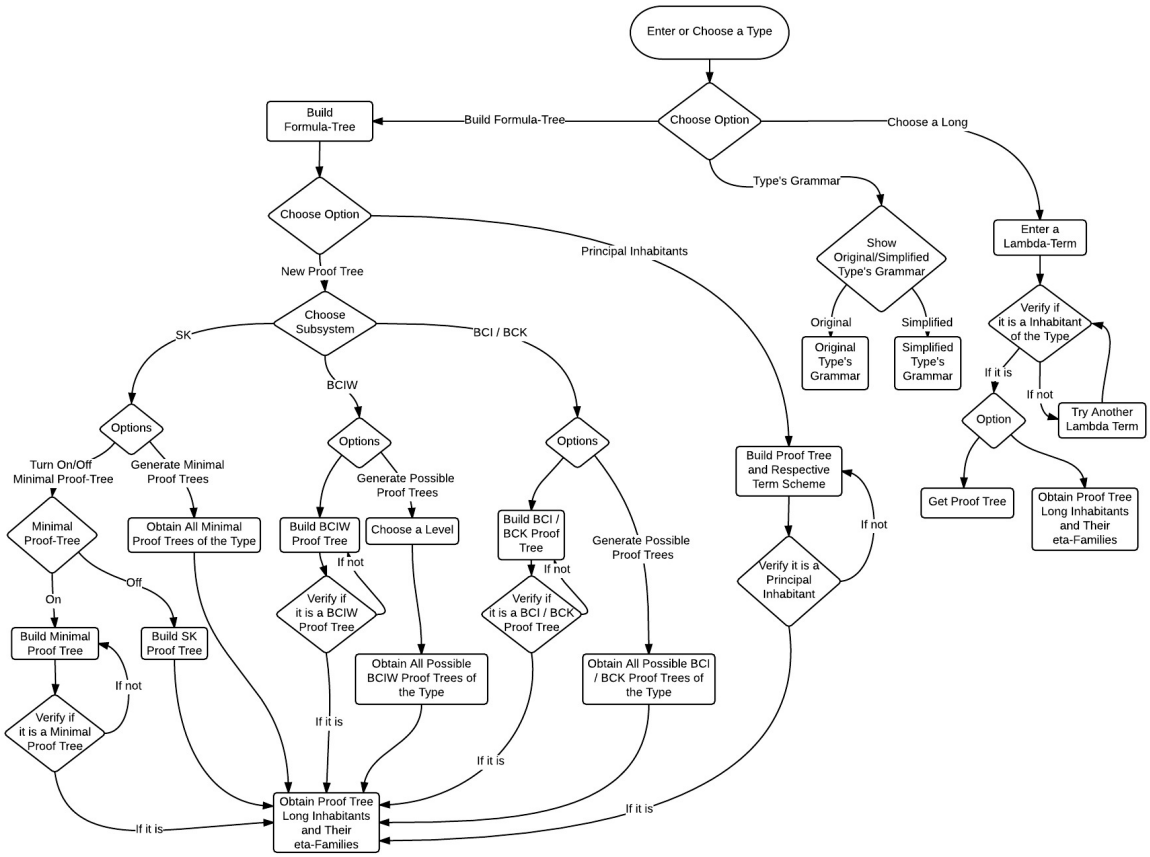


Figure 4.1: Tool Flowchart

4.2 Technologies

We now give a brief description of the used technologies and the main reasons for using such technologies.
The FTM-Tool is a multi-platform web application that works uniformly in the most used web browsers. The application was developed using HTML, CSS, jQuery (a JavaScript library), and PHP. The tool follows a client-server model, although most of the computation is done on the client side. The server is only needed for store the types given to the tool so they can be used again in the future. For the implementation of the features on the client-side we used jQuery, HTML, and CSS. For the feature on the server-side we used PHP. To store the information about the type given to the tool, and to create formula-trees and proof-trees we used JSON objects. In the construction of the tool, more specifically in the structure of the application and in the implementation of the trees, we used two open source tools, Bootstrap [5] and D3.js [19], respectively.

JavaScript
JavaScript is one of the most widely used technologies in the development of web sites and web applications. It has several advantages: is supported by every popular web browser, and it runs locally in the user web browsers, which allows the application to respond quickly to the user actions. Within JavaScript we chose to use jQuery, because it contains additional features that simplify, for example, the selection and handling of HTML elements and CSS manipulation. HTML and CSS are the most widely used technologies in the construction and setting of styles to the web pages, respectively.

JSON Objects
We used JSON objects to store the information about the type given to the tool, and to store all the information necessary to generate formula-trees and proof-trees. JSON is an object notation in JavaScript that provides a compact way to store information, which accelerates the parsing of that information. The JSON objects are constructed from pairs of the form name/value, which can be grouped to form more complex structures, such as arrays of pairs name/value or arrays of JSON objects. Thus, we can say that a JSON object can represent virtually any kind of information.

PHP
We used PHP to store the types given to the tool in a file, so that those types can be used again in the future. JavaScript does not allow us to do these kind of things, so

we needed to find an alternative, and we decided to use PHP, which is a server-side scripting language used for web development. We had other options, such as using Node.js, which has the same purpose of PHP, to construct the server, and MySQL, which is an open-source relational database management system, to save the types in a database. But the small amount of data that we want to save does not justify the creation of a database. Thus, since this problem could be solved using PHP, we opted for this technology.

Bootstrap

Bootstrap is used for designing web sites, and web applications. One of the reasons for the use of Bootstrap was the speed of development. Instead of coding the website from scratch, this tool enables us to use ready made blocks of code. Another advantage of this technology is that it has a fluid grid layout that dynamically adjusts to the proper screen resolution, meaning that the user can use the FTM-Tool in a computer, in a tablet, or even in a smartphone (although smartphones are not the most suitable option, due to the size of the data generated by the application). It is also compatible with the latest versions of the most popular web browsers.

D3.js

To produce interactive visualization of the formula-trees and the proof-trees of the application we used D3.js, which is a JavaScript library. This technology allows us to have access to the information of each node and manipulate it as we want, for example, add/remove primitive parts to the proof-trees. Amongst the various options we considered and tried, D3.js was the one that allowed us to create and manipulate trees dynamically as required. The other options, like Graphviz [22] or even creating a diagram in HTML, only allowed us to create static trees. We used Cluster Dendrogram [21] from D3.js, mainly for aesthetic reasons, but we needed to modify it because we wanted to change the orientation of the tree, from horizontal to vertical, and place the text inside the nodes. We also wanted to change some colors and manipulate the links between nodes, in order to have continuous as well as dashed lines. These changes were made following a tutorial available on a blog [20].

4.3 JSON Objects

In this section we give a description of the JSON objects that store the information of the type given to the tool.
As we have seen before in Chapter 3, types are broken into primitive parts, which are used to build the type's formula-tree, its proof-trees, etc. We need to know specific information about each primitive part. For example: which are the types of its head and tail-variables; which node is its parent; if its tail-variables have descendant primitive parts and, if so, which are they; etc. Thus, it is necessary to save this information, so that it can be accessed and manipulated by the tool whenever necessary. We decided to store this information in two different JSON objects, which are the objects in Example 4.3.1.
To build these objects, the tool starts by breaking the type into subpremises (underlined in the example) to obtain all the nodes and primitive parts. Then it sets the dependencies between nodes and primitive parts, assigns an id to each node, and a name to each primitive part. All this is achieved by following the steps of the formula-tree construction, defined in the previous chapter.
The first object, named treeJSON, is a list. Every element of the list represents a node of the tree, and it has four pairs of the form name/value. The first pair, of each element of the list, represents the id assigned to each node of α, the second pair represents the type of each node, the third pair represents the parent node of each node, and the value of the last pair is a list in which each element contains two pairs of the form name/value with the id, and the type of each child of the node.
The second object, named partsJSON, is a list too. Every element of the list represents a primitive part of the tree, and it has five pairs of the form name/value. The first pair represents the name of the primitive part, the second pair represents the id assigned to the head node of each primitive part, the third pair represents the type of the head node of each primitive part, the fourth pair represents the parent node of the head node of each primitive part, and the last pair is also a list in which each element contains two pairs of the form name value with the id, and the type of each leaf node of each primitive part.
This information is the basis of the entire application and these two objects will be propagated and used throughout the rest of this thesis.

Example 4.3.1. *Consider the type* $\alpha = \underline{\underline{((\underline{a} \to b) \to \underline{a} \to b)}} \to \underline{(\underline{a} \to b)} \to \underline{a} \to b$. *The* treeJSON *object obtained is the one in Figure 4.2, and the* partsJSON *object obtained is the one in figure 4.3.*

```
{"tree":[
{"id":"0","node":"b","parent":"-1","children":[
        {"id":"1","node":"b"},
        {"id":"2","node":"b"},
        {"id":"3","node":"a"}]},
{"id":"1","node":"b","parent":"0","children":[
        {"id":"1_1","node":"b"},
        {"id":"1_2","node":"a"}]},
{"id":"1_1","node":"b","parent":"1","children":[
        {"id":"1_1_1","node":"a"}]},
{"id":"1_1_1","node":"a","parent":"1_1","children":""},
{"id":"1_2","node":"a","parent":"1","children":""},
{"id":"2","node":"b","parent":"0","children":[
        {"id":"2_1","node":"a"}]},
{"id":"2_1","node":"a","parent":"2","children":""},
{"id":"3","node":"a","parent":"0","children":""}]}
```


Figure 4.2: treeJSON Object


```
{"parts":[
{"name":"X1","id":"1","node":"b","parent":"0","children":[
        {"id":"1_1","node":"b"},
        {"id":"1_2","node":"a"}]},
{"name":"X2","id":"2","node":"b","parent":"0","children":[
        {"id":"2_1","node":"a"}]},
{"name":"X3","id":"3","node":"a","parent":"0","children":""},
{"name":"X4","id":"1_1_1","node":"a","parent":"1_1","children":""}]}
```


Figure 4.3: partsJSON Object

4.4 Building The Formula-Tree Of The Type

In Chapter 3 we explained the method for building the formula-tree. In this section we will show how this method was implemented.

To implement the option **Build Formula-Tree** it is necessary to construct a new JSON object, named dataJSON. This dataJSON object is constructed using the JSON objects from Section 4.3, and is used to store the information about the formula-tree. This information is passed to D3.js in order to display the formula-tree of the type on the web browser.

This dataJSON object (Figure 4.4) is a list, where each element represents a node of the tree, composed of four pairs of the form name/value. The first pair, of each element of the list, represents the id assigned to each node of α; the second pair represents the type of each node; the third pair represents the parent node of each node; and the value of the last pair defines the type of line between the node and its parent node (continuous or dashed). To build this new object, dataJSON, it is necessary to go through each element of the treeJSON object and add it to the dataJSON object, with its respective id and parent. Each time a new element is added to the dataJSON object

it is necessary to calculate if the line of that element is continuous or dashed, for which the tool uses the partsJSON object. If the object is the head of a primitive part, then its line is set as dashed, otherwise its line is set as continuous. Note that the first node of the object is a blank element (an element in which the name is empty), and the purpose of this element is to obtain the first horizontal line of the proof-tree. We have other blank elements that are piece of the primitive parts that have no tail-variables, i.e. the leaf nodes, the primitive parts that close the proof-trees.
In the following we will see an example of a dataJSON, and its resultant formula-tree.

Example 4.4.1. *Consider α, and the* treeJSON *and* partsJSON *objects from Example 4.3.1. The following* dataJSON *object is obtained:*

```
[{ "id":"-1", "name" : "", "parent":"null", "line":"continuous" },
{ "id":"0", "name" : "b", "parent":"-1", "line":"continuous" },
{ "id":"1", "name" : "b", "parent":"0", "line":"dashed" },
{ "id":"1_1", "name" : "b", "parent":"1", "line":"continuous" },
{ "id":"1_1_1", "name" : "a", "parent":"1_1", "line":"dashed" },
{ "id":"1_1_1_1", "name" : "", "parent":"1_1_1", "line":"continuous" },
{ "id":"1_2", "name" : "a", "parent":"1", "line":"continuous" },
{ "id":"2", "name" : "b", "parent":"0", "line":"dashed" },
{ "id":"2_1", "name" : "a", "parent":"2", "line":"continuous" },
{ "id":"3", "name" : "a", "parent":"0", "line":"dashed" },
{ "id":"3_1", "name" : "", "parent":"3", "line":"continuous" }]
```

Figure 4.4: dataJSON Object

The resultant formula-tree of α is:

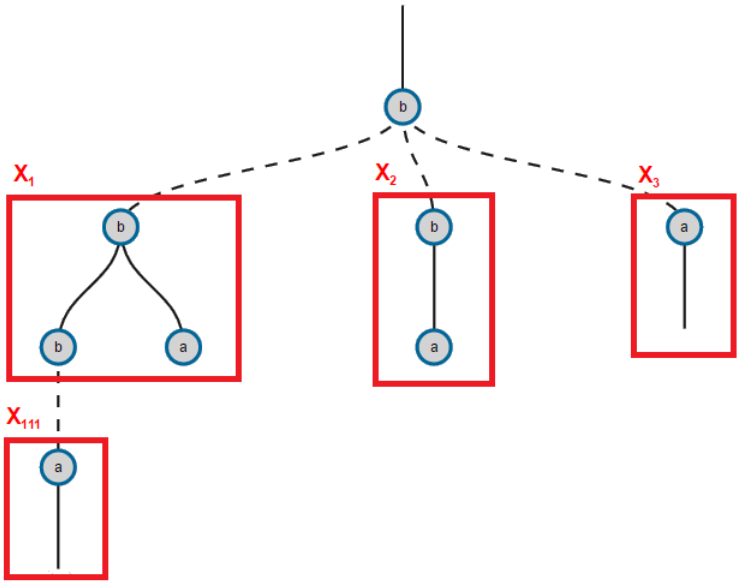


Figure 4.5: Formula-Tree of α

4.4.1 Building Proof-Trees

One of the most important features of this tool is the construction of proof-trees. This can be done interactively, after clicking on the option **Build proof-trees**, or automatically as we will see in Section 4.4.1.1.
Recall from the formula-tree method that the available primitive parts can be used as many times as we want. However, since there is an hierarchy defined between the primitive parts, some of them might not be available at all steps.
To implement the interactive generation of proof-trees, we have to create a new JSON object, called dataJSON. This dataJSON object is different from the one we saw in Section 4.4.
In the following we will see an example of this new dataJSON object.

Example 4.4.2. *Consider* $\alpha = ((a \to b) \to a \to b) \to (a \to b) \to a \to b$*, as before, its formula-tree in Figure 4.5, and the proof-tree in Figure 4.6. The corresponding* dataJSON *object is shown in Figure 4.7.*

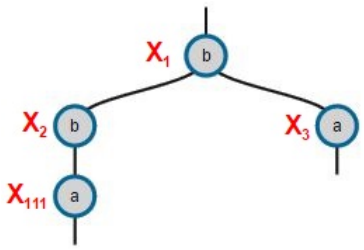


Figure 4.6: Proof-Tree of α


```
[{"id":"-1","idP":"-1","name":"","parent":"null","previous":"null","add":"no","rm":"no","lvl":"-1"},
 {"id":"1","idP":"1","name":"b","parent":"-1","previous":"0,0,-1,0","add":"no","rm":"yes","piece":"X1",
  "lvl":"0.1","history":"{x1x2x3}."},
 {"id":"15","idP":"2","name":"b","parent":"1","previous":"2,1_1,1,0.1.1_1","add":"no","rm":"yes","piece":"X2",
  "lvl":"0.1.1_1.2","history":"{x1x2x3}.x1.{x1x2x3x1_1_1}."},
 {"id":"13","idP":"3","name":"a","parent":"1","previous":"3,1_2,1,0.1.1_2","add":"no","rm":"yes","piece":"X3",
  "lvl":"0.1.1_2.3","history":"{x1x2x3}.x1.{x1x2x3}."},
 {"id":"14","idP":"3aux","name":"","parent":"13","previous":"3,1_2,1,0.1.1_2","add":"yes","rm":"no","piece":"X3",
  "lvl":"0.1.1_2.3.3aux","history":"{x1x2x3}.x1.{x1x2x3}.x3.{x1x2x3}."},
 {"id":"17","idP":"1_1_1","name":"a","parent":"15","previous":"16,2_1,15,0.1.1_1.2.2_1","add":"no","rm":"yes",
  "piece":"X1_1_1","lvl":"0.1.1_1.2.2_1.1_1_1","history":"{x1x2x3}.x1.{x1x2x3x1_1_1}.x2.{x1x2x3x1_1_1}."},
 {"id":"18","idP":"1_1_1aux","name":"","parent":"17","previous":"16,2_1,15,0.1.1_1.2.2_1","add":"yes","rm":"no",
  "piece":"X1_1_1","lvl":"0.1.1_1.2.2_1.1_1_1.1_1_1aux",
  "history":"{x1x2x3}.x1.{x1x2x3x1_1_1}.x2.{x1x2x3x1_1_1}.x1_1_1.{x1x2x3x1_1_1}."}]
```

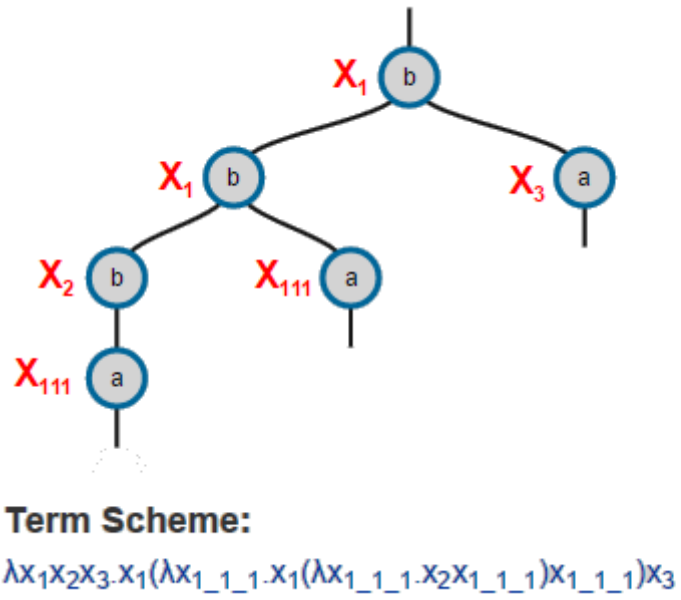

Figure 4.7: dataJSON Object

This dataJSON *object is a list too. Every element of the list represents a node of the proof-tree, but it has now seven pairs of the form name/value instead of four. For each node, i.e. element of the list, the first pair represents the id assigned to the node; the second pair represents the id assigned to the primitive part to which the node belongs*

to; and the third pair represents the parent node of the node (idP). *The fourth pair indicates if is possible to add a new primitive part to the node* (add), *and the fifth pair indicates if it is possible to remove the node (and consequently the sub-tree below it)* (rm). *The last two pairs represent the depth of the node in the proof-tree* (lvl), *and the context in which each primitive part was used along the path* (history), *respectively. The context tells us which are the available pieces at a given time, and will be necessary during the construction of minimal proof-trees.*
At the beginning of the proof-tree construction this object only contains two elements: a null element and the root node of the formula-tree. Everytime a primitive part is added to the proof-tree, the node where the primitive part was added is replaced in the dataJSON *object by the head-variable of the primitive part. The head-variable inherits the* idP *of the node that it is replacing, and the* lvl, add, rm, *and* history *fields are updated. Then, the tail-variables are added to the* dataJSON *object.*

The treeJSON and partsJSON objects are used to obtain the available primitive parts for each node, so there is no risk of the user adding a primitive part in the wrong place.
In the following we will see an example of two possible proof-trees.

Example 4.4.3. *Consider* $\alpha = ((a \to b) \to a \to b) \to (a \to b) \to a \to b$.
Any one of the proof-trees in Figure 4.8 can be obtained.



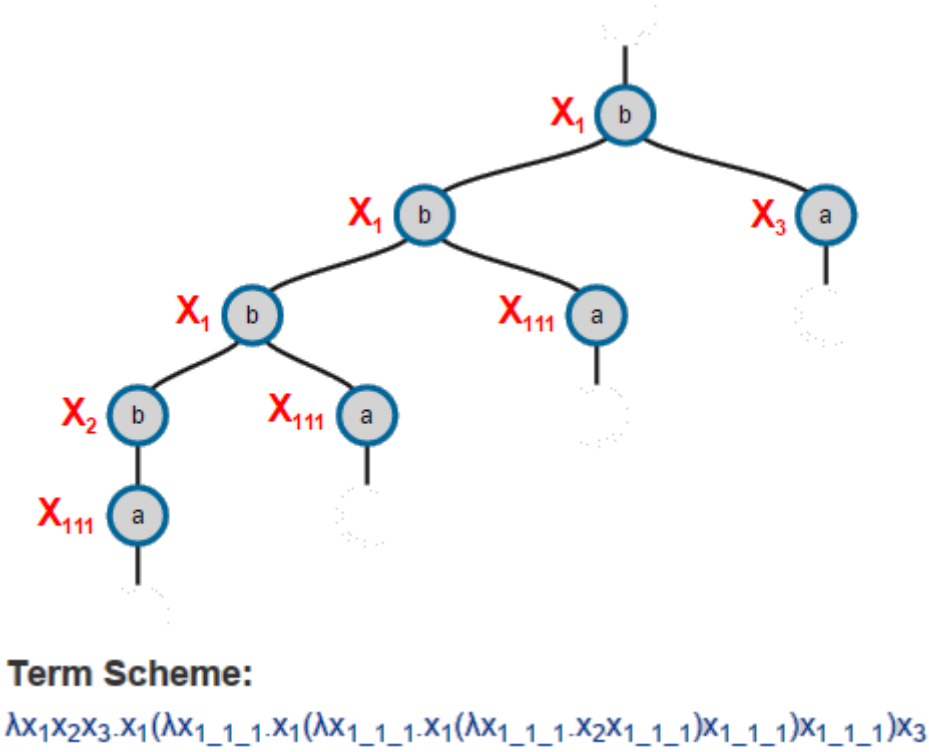


Figure 4.8: Proof-Trees Example for α

Note that in this section there are no restrictions on the λ-terms, since we are in the complete system, i.e. the **SK**-calculus.

4.4.1.1 Minimal Proof-Trees

The construction of minimal proof-trees is particularly useful, if one is interested in deciding about the existence of a long inhabitant for a type, or equivalently, in deciding if a formula is a theorem in the implicational fragment of intuitionistic propositional logic. To implement the construction of minimal proof-trees, we applied some restrictions on the construction of proof-trees in the **SK**-calculus. Note that, when a primitive part is added to a proof-tree, it has a context (a set of available primitive parts) associated. Now, suppose that during the construction of a proof-tree the same primitive part is added twice in a same branch and both times the associated context is the same, if in the end it is possible to close the proof-tree, this means that it was possible to close the proof-tree the first time the primitive part was added. Meaning that there is no need to add a piece in the same context twice.
In the following example we will see the construction of a minimal proof-tree, which is a proof-tree in which each primitive part can only be added once in each context in the same branch of a proof-tree.

Example 4.4.4. *Consider α from the previous examples and its formula-tree from Figure 4.5. Figure 4.9 shows two proof-trees. The first one is a minimal proof-tree.*

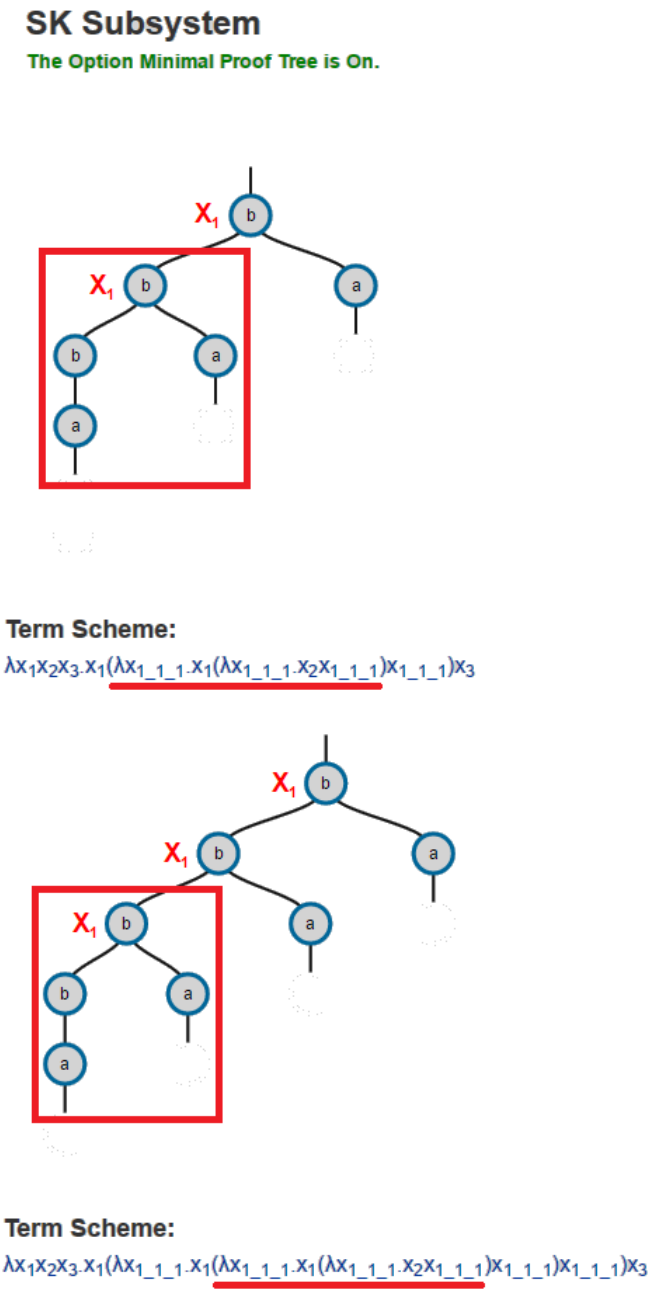


Figure 4.9: SK Proof-Tree vs Minimal Proof-Tree

However, looking at Figure 4.9, it is possible to see that in the second proof-tree the primitive part X_1 was added three times in the left branch. The first time the primitive parts X_1, X_2, and X_3 were available, i.e. the associated context was X_1, X_2, X_3. When X_1 was added for the second time X_1, X_2, X_3, and X_{111} were available. When X_1

was added for the third, the available pieces remained the same, which means that the context was repeated. Thus, the second proof-tree is not a minimal proof-tree.

The minimal proof-trees implementation is the same as the implementation of proof-trees that we have seen in Section 4.4.1, but for the verification of the context in which a primitive part is used. This verification is done by adding to each added primitive part, a parameter history. This parameter stores the history of the path in which the primitive part was added, i.e. keeps the contexts in which each piece was added along the branch. When a minimal proof-tree is being constructed, the tool uses the history to verify if a primitive part was already used in that context. If this is the case, that primitive part will no longer be available to add.

4.4.1.2 Generate Minimal Proof-Trees

Since there are context restrictions on the generation of minimal proof-trees, the number of proof-trees that can be built is limited. The maximum depth of a minimal proof-tree is $n \times 2^n$, where n is the number of primitive parts of the type. Thus, it is possible to generate all minimal proof-trees for a type.
If the option **Generate Minimal proof-trees** is chosen, the tool will generate automatically all minimal proof-trees, and will calculate the number of inhabitants for the chosen type.
The generation of the minimal proof-trees is done by running through the nodes of the proof-tree and verifying if it is possible to add a primitive part to a node. If it is, the available pieces for that node are obtained, and it is verified if those primitive parts have not already been added in the same context. If at the end of the context verification there is only one option available, the new primitive part is added to the tree, but if there is more than one option, the tree is copied the necessary number of times, and a different primitive part is added to each one of the resulting trees. If at some point there are no options for a node, that tree is discarded. This information is stored in a different dataJSON object for each generated proof-tree.
To calculate the number of inhabitants of the type, the first thing that the tool does is to verify if the number of inhabitants is finite or infinite. If in at least one minimal proof-tree there are two primitive parts (in the same branch) whose head-variable have the same type-variable, the number of inhabitants is infinite. If there are no primitive parts whose head-variables have the same type-variable (in the same branch), then it is necessary to calculate the long inhabitants and their η-families of each one of the possible minimal proof-trees and count them. If the type has no minimal proof-trees, then the number of inhabitants is zero. Note that, if we do not set the context

restriction, the same primitive part could always be added in the same context, and this would generate an infinite number of trees, and consequently, an infinite number of long inhabitants.
In the following we will see an example of the result of the automatic generation of minimal proof-trees.

Example 4.4.5. *Given α from the previous examples and considering its formula-tree from Figure 4.5, the following proof-trees are obtained:*

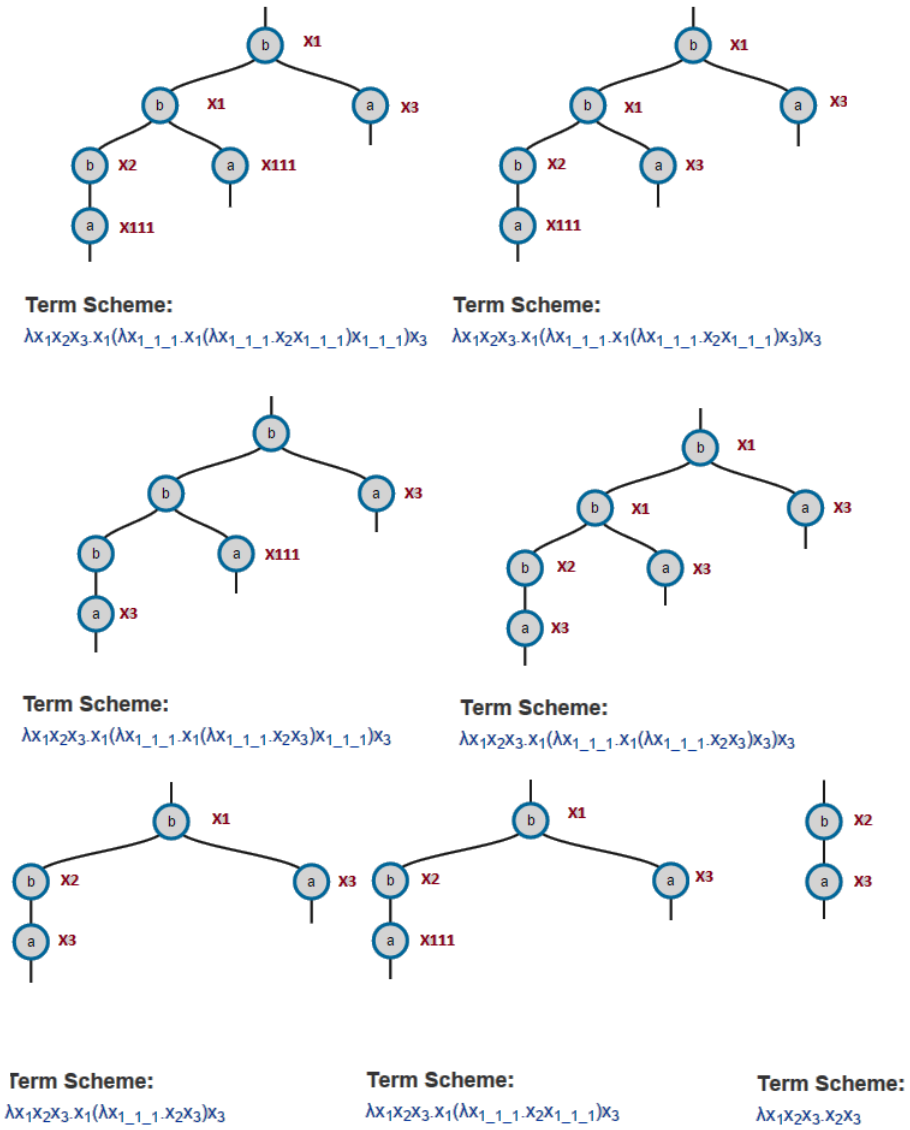


Figure 4.10: Minimal Proof-Trees

The type α has an infinite number of inhabitants.

4.5 Proof-Tree Long Inhabitants And Their η-Families

Another important feature of the FTM-Tool is the computation of the long inhabitants and their η-families, from the proof-trees built or automatically generated by the tool. To implement this feature, the first thing that needs to be done is to verify if it is necessary to change the name of any abstraction and variables (remember Section 3.3). To do that, the tool verifies if the term-scheme has different lambdas with the same name. In that case, the tool adds a number to each lambda in order to distinguish them, and the variables are renamed in all possible ways, by considering all possible alternatives in a given context. Once the long inhabitants are calculated, the tool can calculate their correspondent η-families by η-reduction. To do that, the tool has to apply recursively η-reduction to each one of the long inhabitants.
We illustrate this process with an example.

Example 4.5.1. *Consider* $\alpha = ((a \to b) \to a \to b) \to (a \to b) \to a \to b$ *and Figure 4.8.*
The resultant term-scheme is:

$$\lambda x_1 x_2 x_3 . x_1 (\lambda x_{111} . x_1 (\lambda x_{111} . x_3 x_{111}) x_{111}) x_3$$

Let us start by renaming the lambdas with the same name:

$$\lambda x_1 x_2 x_3 . x_1 (\lambda x_{111[1]} . x_1 (\lambda x_{111[2]} . x_3 x_{111}) x_{111}) x_3$$

We can now see that the rightmost x_{111} *needs to be renamed to* $x_{111[1]}$*, but the leftmost* x_{111} *can either be* $x_{111[1]}$ *or* $x_{111[2]}$*. This results in the following two long inhabitants:*

$$\lambda x_1 x_2 x_3 . x_1 (\lambda x_{111[1]} . x_1 (\lambda x_{111[2]} . x_3 x_{111[1]}) x_{111[1]}) x_3$$

$$\lambda x_1 x_2 x_3 . x_1 (\lambda x_{111[1]} . x_1 (\lambda x_{111[2]} . x_3 x_{111[2]}) x_{111[1]}) x_3$$

After the η*-reductions, the tool computes the following long inhabitants and their* η*-families:*

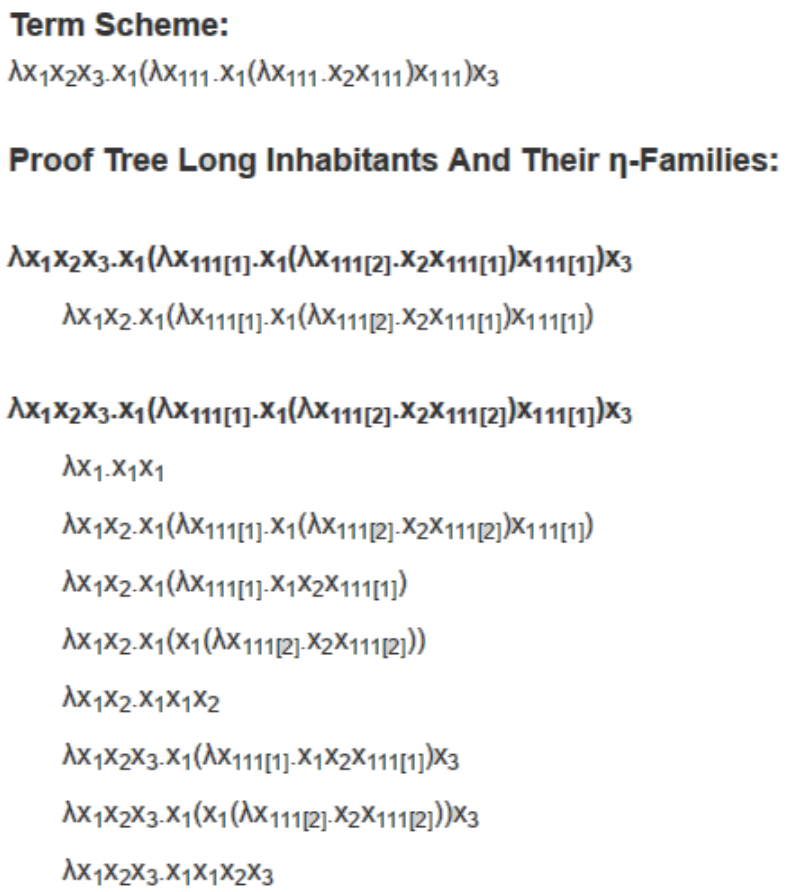


Figure 4.11: Calculating Long Inhabitants and their η-Families

4.6 Subsystems

In this chapter we have already seen the construction of proof-trees for the entire **SK**-calculus. Additionally, the FTM-Tool has the option to build proof-trees in three different subsystems, namely the **BCIW**-, **BCI**-, and **BCK**-subsystems.
In all of these three subsystems, the proof-trees are constructed in the same way as described in Section 4.4, but for each one of them there are different restrictions. The restrictions of each subsystem will be presented in the following subsections, as well as the implementation details for each subsystem.

4.6.1 BCIW Proof-Trees

As we saw in Section 2.3, the restriction in **BCIW** is that, if a λ-term is of the form $\lambda x.M$, then x has to occur free in M at least once. In particular, all the primitive parts have to be used at least once.

The tool can confirm if the proof-tree is being correctly constructed by running a parser on the term-scheme everytime a new leaf node is added to the proof-tree. This verifies if in every subterm (of the term-scheme) of the form $\lambda x.M$, x occurs free in M at least once. For example, if the term-scheme is $\lambda x_1 x_2 x_3.x_1(\lambda x_{111}.x_2 x_3)\Delta$, it is possible to see, even while the proof-tree is not closed, that it is not correct because inside of the λx_{111} there is no x_{111}. In this case, an error message is obtained as soon as the primitive part x_3 is introduced. The tool has also a flag that verifies if in the end of the proof-tree construction every primitive part was used.
In the following we will see an example of a badly formed and a well formed proof-tree.

Example 4.6.1. *Consider type α from the previous examples and its formula-tree from Figure 4.5.*
Figure 4.12 shows an example of a well formed **BCIW** *proof-tree. On the other hand, as we can see in Figure 4.13, if only primitive parts X_2 and X_3 are added to the proof tree, an error is obtained because there are available primitive parts that were not used.*

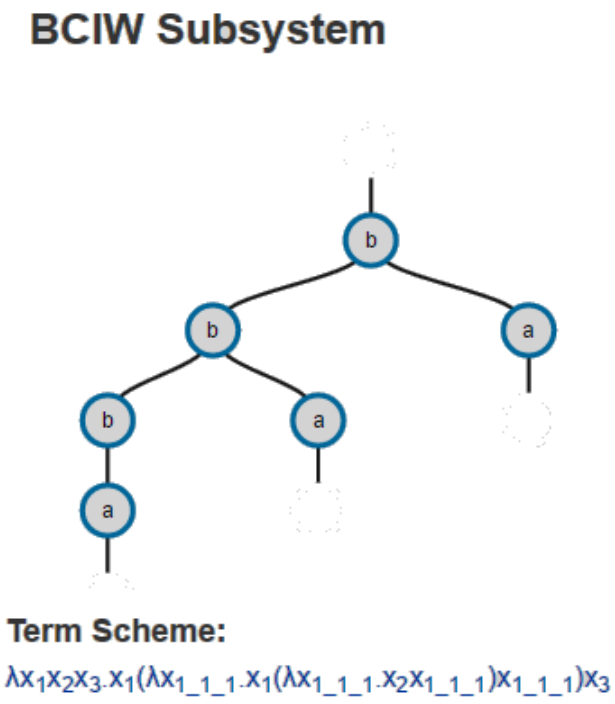


Figure 4.12: Well Formed **BCIW** Proof-Tree

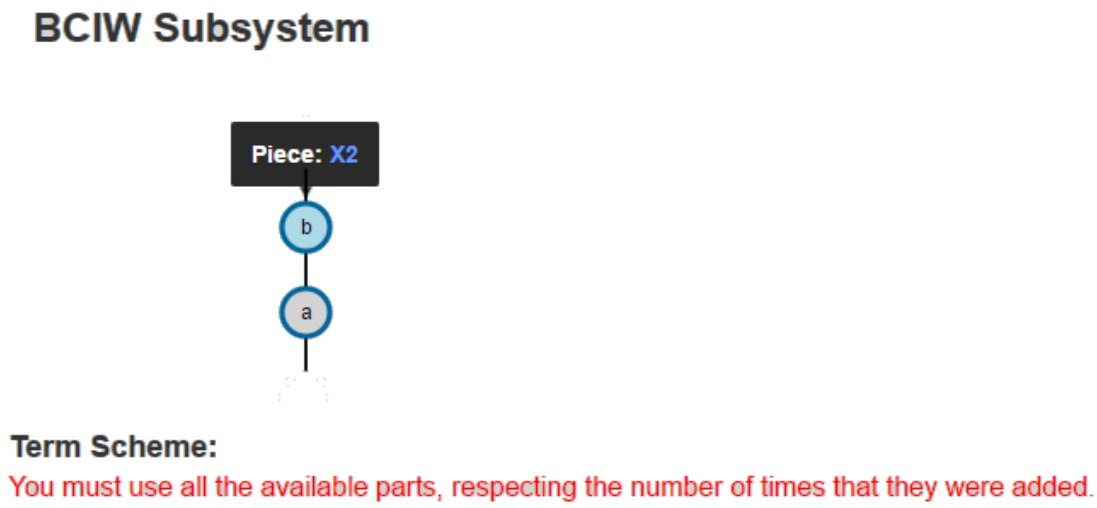
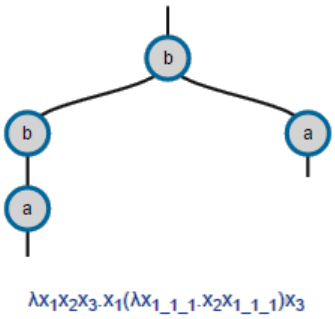


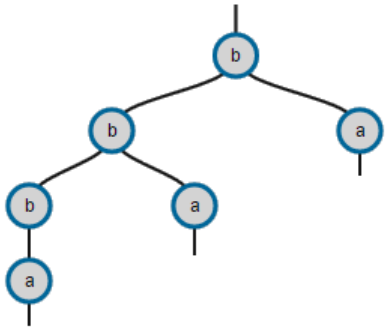
Figure 4.13: BCIW Construction

4.6.1.1 Generate BCIW Proof-Trees

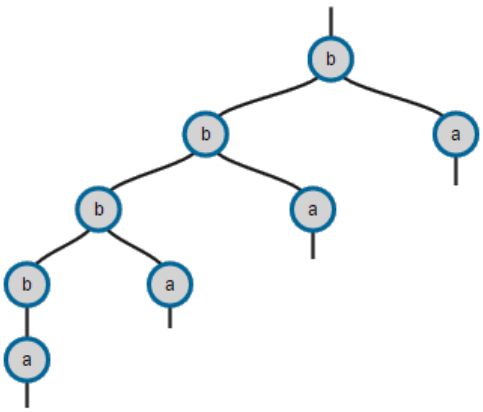
It is possible to automatically generate the proof-trees in the **BCIW** fragment by choosing the option **Generate Possible Proof-Trees**. The halting condition of **SK**-calculus no longer applies, as shown in [13]. Without this restriction the method could generate an infinite number of proof-trees or even run forever without generating any proof-tree. This means that is necessary to give an halting condition, for which we chose maximum depth of the proof-trees generated by the tool. Note that, if the provided depth is too low the tool may not obtain any proof-trees.
In the following we will see an example.

Example 4.6.2. *Consider α from the previous examples. If the chosen depth is to low, such as level 2, no proof-tree will be obtained, but if the chosen depth is 6, the following proof-trees are obtained:*

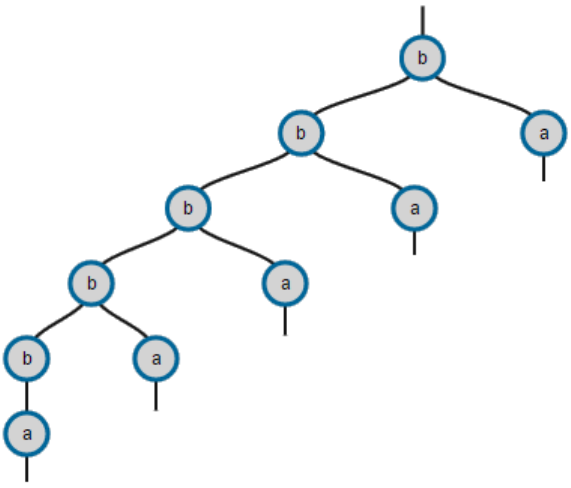




$\lambda x_1 x_2 x_3.x_1(\lambda x_{1\_1\_1}.x_1(\lambda x_{1\_1\_1}.x_2 x_{1\_1\_1})x_{1\_1\_1})x_3$



$\lambda x_1 x_2 x_3.x_1(\lambda x_{1\_1\_1}.x_1(\lambda x_{1\_1\_1}.x_1(\lambda x_{1\_1\_1}.x_2 x_{1\_1\_1})x_{1\_1\_1})x_{1\_1\_1})x_3$



$\lambda x_1 x_2 x_3.x_1(\lambda x_{1\_1\_1}.x_1(\lambda x_{1\_1\_1}.x_1(\lambda x_{1\_1\_1}.x_1(\lambda x_{1\_1\_1}.x_2 x_{1\_1\_1})x_{1\_1\_1})x_{1\_1\_1})x_{1\_1\_1})x_3$

Figure 4.14: BCIW - Level 6

4.6.1.2 Long Inhabitants and Their η-Families in BCIW

The generation of long inhabitants and their η-families in **BCIW** is the same as shown in Section 4.5. However, between the generation of the long inhabitants and the generation of their η-families, an intermediate step was added. When the tool renames the variables in the term-scheme, the resultant long inhabitant might not respect the restriction of the **BCIW**-subsystem. Thus, the tool verifies if x occurs in M at least once for every $\lambda x.M$. All other long inhabitants and their η-families have to be discarded, as they do not belong to the **BCIW**-subsystem.
In the following we will see an example.

Example 4.6.3. *Consider the term-scheme:*

$$\lambda x_1 x_2 x_3.x_1(\lambda x_{111}.x_1(\lambda x_{111} x_2 x_{111})x_{111})x_3$$

And the resultant long inhabitants:

$$\lambda x_1 x_2 x_3.x_1(\lambda x_{111[1]}.x_1(\lambda x_{111[2]}.x_3 x_{111[1]})x_{111[1]})x_3$$

$$\lambda x_1 x_2 x_3.x_1(\lambda x_{111[1]}.x_1(\lambda x_{111[2]}.x_3 x_{111[2]})x_{111[1]})x_3$$

We can see that the first one does not respect the restriction of the **BCIW***, therefore does not belong to the subsystem.*

4.6.2 BCI and BCK Proof-Trees

As we saw in Section 2.3, the restriction in the **BCI** subsystem is that every subterm $\lambda x.M$ has to be such that x occurs exactly once free in M. Consequently, all primitive parts need to be used exactly once in the proof-tree. Furthermore, in the **BCK** subsystem the restriction is that every subterm $\lambda x.M$ has to be such that x occurs at most once free in M. Consequently, each primitive part can be used at most once. This means that it is not necessary to use all the primitive parts; but once a primitive part is used, it can no longer be used again.
We will now see how these restrictions of were implemented. To verify if the built proof-tree is correct we created a structure with each primitive part accompanied by a flag with the number zero. Every time a new primitive part is added to the proof-tree the zero of that primitive part is changed to one. When an entire sub-tree is removed from the proof-tree, it is necessary to verify what primitive parts were removed and change their flag from one to zero. In the **BCI**-subsystem, if at the end of the construction of the proof-tree there is a primitive part whose flag is zero,

then the tool shows an error, because that primitive part has not been used. In the **BCK**-subsystem there is no need to verify if all the primitive parts have been used. There is no risk of adding the same primitive part more than once, because once the primitive part has been used (its flag is one), it is no longer available to be used again.

4.6.2.1 Generate BCI and BCK Possible Proof-Trees

Just as in **BCIW** logic, it is possible to generate proof-trees automatically by chosing the option **Generate Possible Proof-Trees**. Since each primitive part can only be used once, there is no need to set an halting condition, the tool will generate a finite number of proof-trees.
In the generation of the proof-trees, the tool has a structure with each primitive part and its flag. If during the automatic construction of the proof-tree the flag of some primitive part switches to one, then this piece can no longer be added to the proof-tree. The difference between the **BCI**- and the **BCK**-subsystem is that for the first the tool needs to verify if in the end of the proof-tree construction there are primitive parts that have not been used, in which case that proof-tree is discarded.
In the following we will see an example for the **BCI**-subsystem.

Example 4.6.4. *Considering type* $\alpha = ((a \to b) \to a \to b) \to ((a \to b) \to a \to b) \to (a \to b) \to a \to b$*, the following possible proof-trees are obtained:*

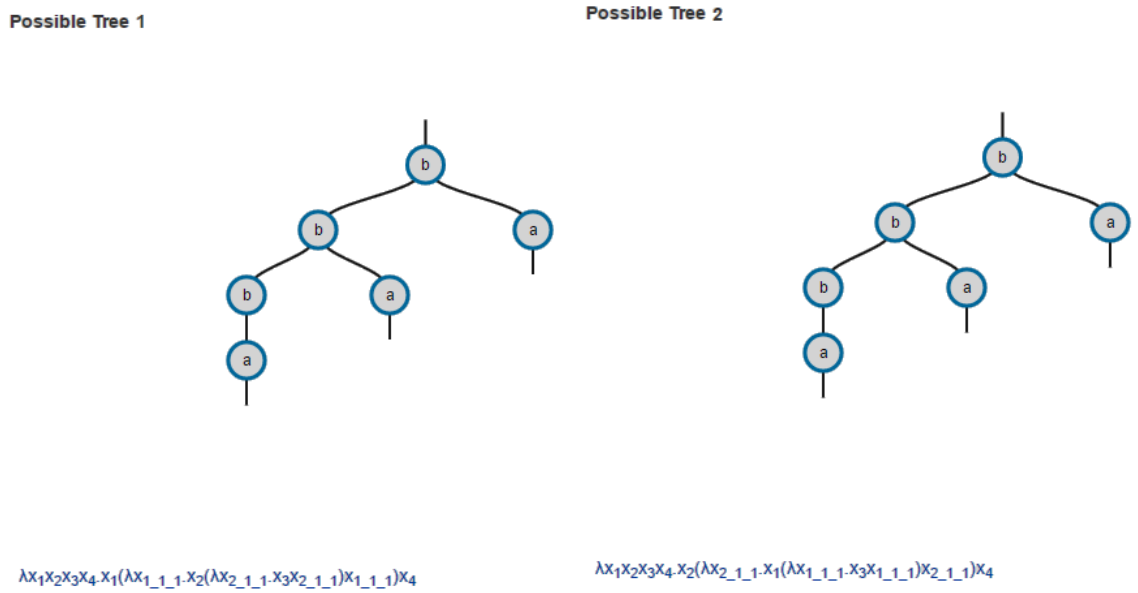


Figure 4.15: BCI - Possible Proof-Trees

The difference between them is the order in which the primitive parts x_1 *and* x_2 *were used.*

In the following we will see an example for the **BCK**-subsystem.

Example 4.6.5. *Having* $\alpha = ((a \to b) \to a \to b) \to ((a \to b) \to a \to b) \to (a \to b) \to a \to b$*, we obtain the following possible trees:*

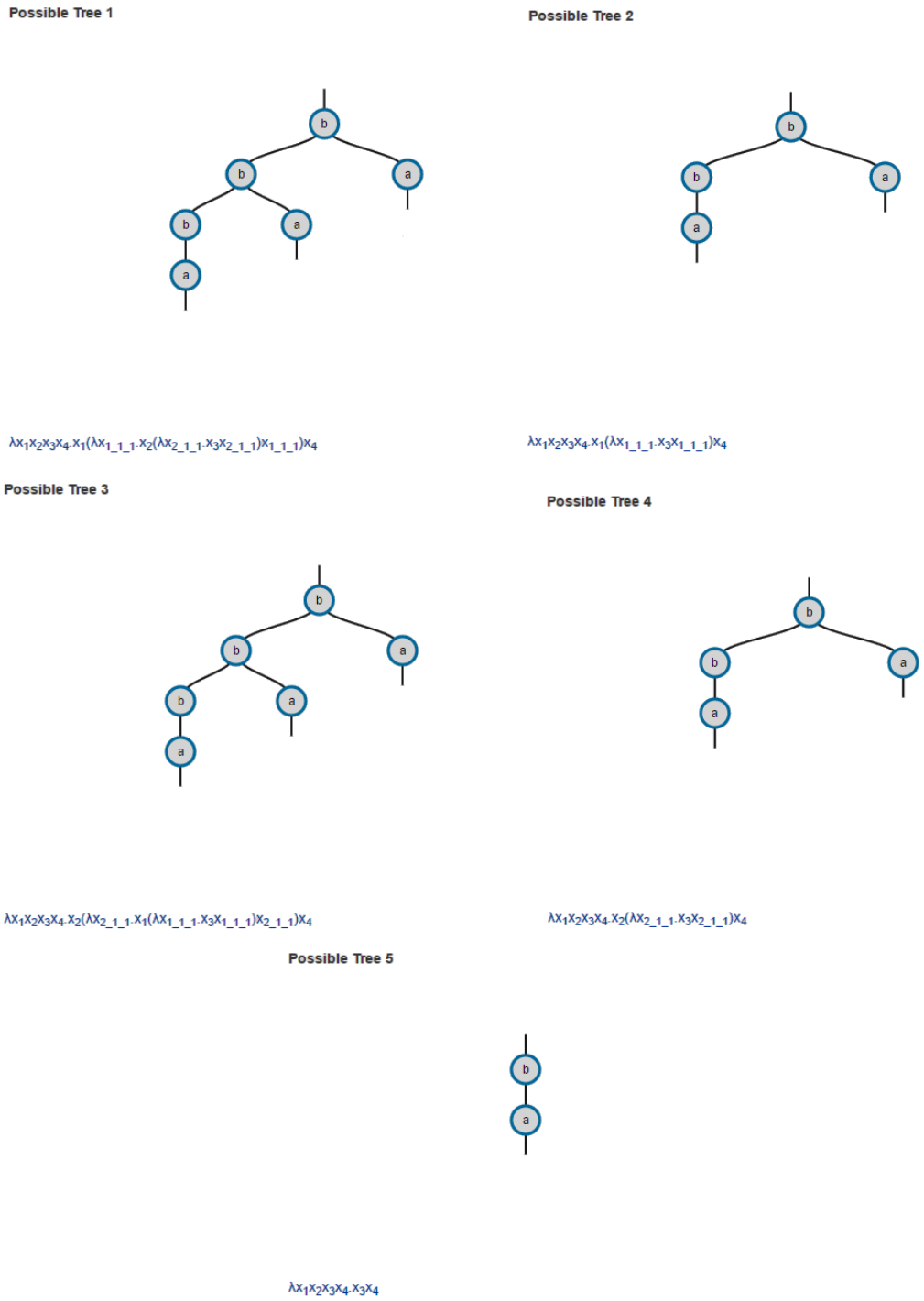


Figure 4.16: BCK - Possible Proof-Trees

Chapter 5

Other Features

In the previous chapter we described the main features of the FTM-Tool. In this chapter we will see other features that were implemented in the tool. Namely, we will detail what is necessary for an inhabitant to be principal. We will also explain how this feature was implemented, and how the tool generates a context-free grammar for a type, from which all its normal inhabitants can be obtained. We will also explain how the tool verifies if a λ-term inhabits a type, and how it builds its proof-tree, and computes its long inhabitants and their η-family.

5.1 Principal Inhabitants

After the tool builds the formula-tree of a type α, the option **Principal Inhabitants** is available. This option will provide support in the construction of a proof-tree for which the set of long inhabitants are principal inhabitants of α.
For an inhabitant to be principal, it is necessary that occurrences of type variables are the same if and only if, this is required by the structure of the inhabitant, and that every arrow in α is necessary. Note that arrows in α corresponding to a dashed line in the formula-tree, i.e. joining the l^{th} tail-variable of a primitive part p_i, with the head-variable of a primitive part p_j are justified if p_i is used, since in this case there will be some occurrence of variable x_i in M with an abstraction λx_j in the l^{th} argument. The other arrows represent the links between the head-variable and the tail-variables in primitive parts. Everytime a primitive part is added to the proof-tree, since the number of tail-variables and the number of arguments added to the term-scheme are the same, the corresponding arrows in α are in fact necessary, i.e. justified. While the proof-tree is being constructed, the head-variable of a primitive part is being overlapped with the tail-variable of another primitive part. This forces these

occurrences of type-variables to be the same in any type of an inhabitant corresponding to this proof-tree. In the end of the proof-tree construction, there has to be enough overlapping of type-variables to assure that all occurrences of the same type-variable need to be identical. To guarantee this, we store this information in equivalence classes. Initially each occurrence of a type variable in the formula-tree has its own equivalence class. When two occurrences of a type variable in the proof-tree are overlapped, the equivalence classes of those occurrences will be merged. In the end of the proof-tree construction, all occurrences of the same atomic type need to be in the same equivalence class.
In the following example we can see the construction of a proof-tree whose associated long inhabitants are principal.

Example 5.1.1. *Consider type* $\alpha = ((a \to b) \to a \to b) \to (a \to b) \to a \to b$ *as before.*
In the beginning of the proof-tree creation the following formula-tree will be shown:

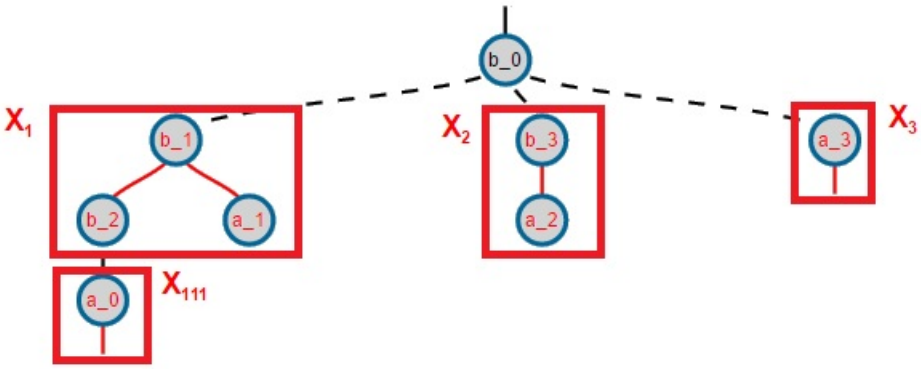


Figure 5.1: Principal Inhabitants Formula-Tree

At this point the lines and the nodes in the primitive parts X_1, X_2 *and* X_3 *are depicted in red, since none of the parts has been used yet.*
Next to the proof-tree, the information about the current equivalence classes is available. In this example there are only two atomic types, a *and* b*. This means that at the end of the proof-tree construction there should be exactly two equivalence classes. One with all the elements whose atomic type is* b *and one with all the elements*

whose atomic type is a. In the beginning of the proof tree construction the equivalence classes are the following:

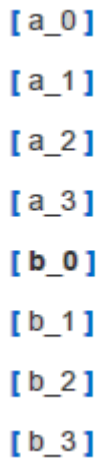


Figure 5.2: Equivalence Classes

If primitive part X_1 is added to the proof-tree, then node b_0 is overlapped with node b_1. Consequently their equivalence classes are merged. After adding X_1 to the proof-tree, there are changes in the colors of some nodes and edges in the formula-tree. In particular, the lines of X_1 are now black, indicating that this primitive part has already been used.

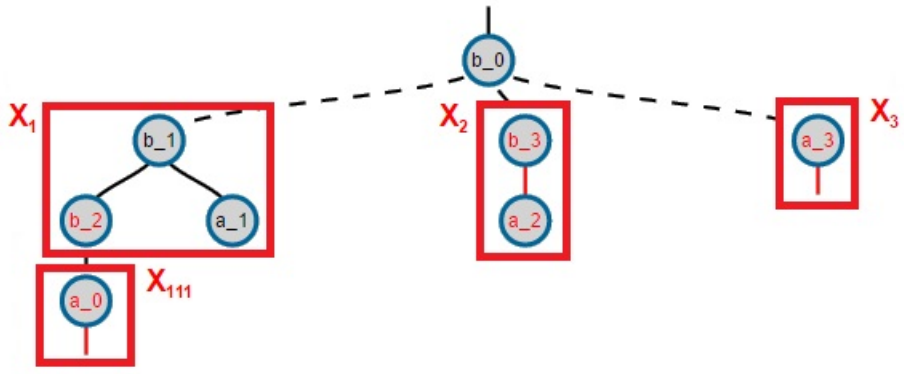


Figure 5.3: Principal Inhabitants Formula-Tree

Continuing this process, after reaching for instance the proof-tree in Figure 5.4 all lines and all nodes of the primitive parts of the formula-tree on Figure 5.5 are now black and there are exactly two equivalence classes, $[b\_0, b\_1, b\_2, b\_3]$ *and* $[a\_0, a\_1, a\_2, a\_3]$. *This means that the long inhabitants corresponding to this proof-tree are principal inhabitants of* α.

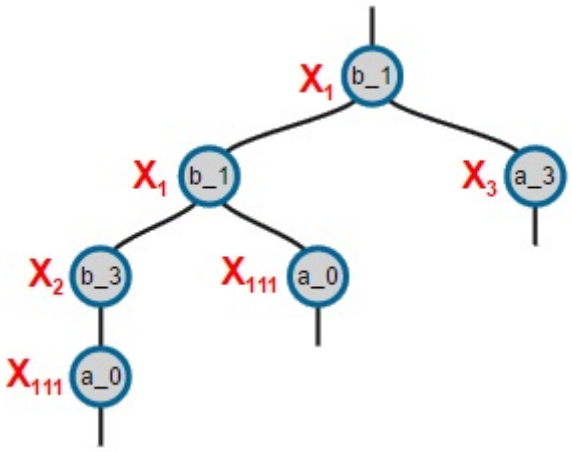


Figure 5.4: Principal Proof-Tree

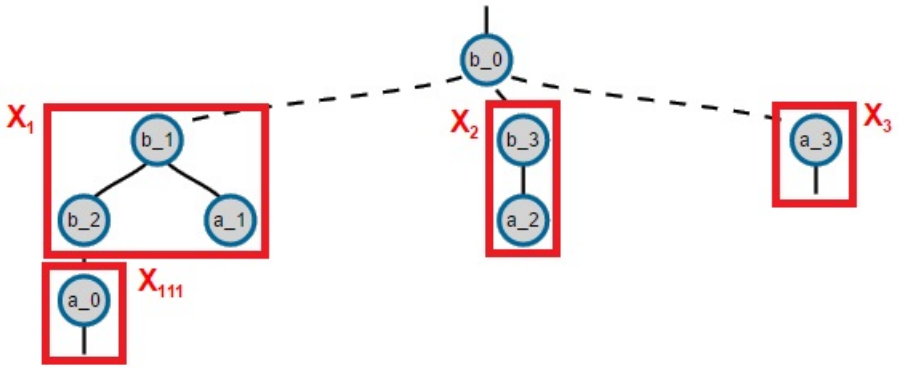


Figure 5.5: Final Formula-Tree

As previously stated, the tool provides support in the construction of a proof-tree for which the set of long inhabitants are principal inhabitants, but does not prevent the creation of other proof-trees. In this case, the tool provides the option to compute the

principal type (represented by its formula-tree) of the long inhabitants that correspond to the constructed proof-tree.
In the following we will see an example:

Example 5.1.2. *Consider α from the previous example and the following proof-tree and equivalence classes:*

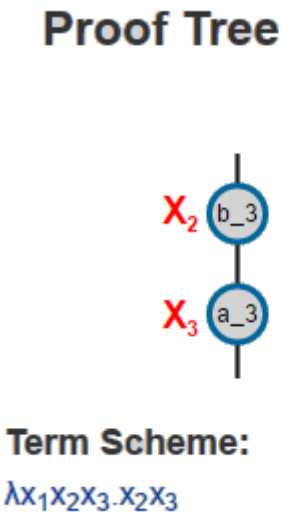


Figure 5.6: Proof-Tree

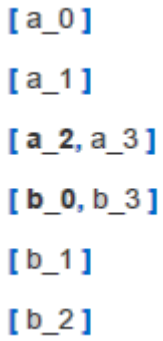


Figure 5.7: Proof-Tree Equivalence Classes

And the following formula-tree:

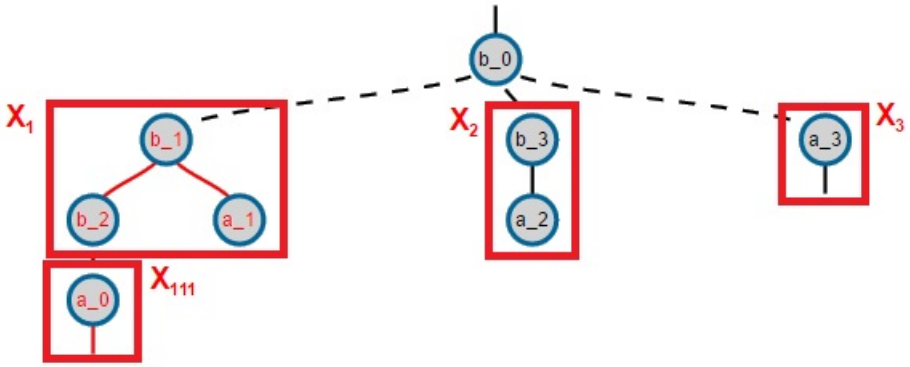


Figure 5.8: Formula-Tree

Primitive parts X_1 and X_{111} have not been use, and their lines and nodes are still red. Thus, this is not a proof-tree for which the set of long inhabitants are principal

inhabitants. This means that the long inhabitant corresponding to this proof-tree, i.e. the term $\lambda x_1 x_2 x 3_. x_2 x_3$*, is no principal inhabitant of* α*.*
If option **Show The Formula-Tree of the Principal Type** *is chosen for the proof-tree in Figure* **??***, then the principal type of the term* $\lambda x_1 x_2 x_3 . x_2 x_3$ *is shown, cf. Figure 5.9.*

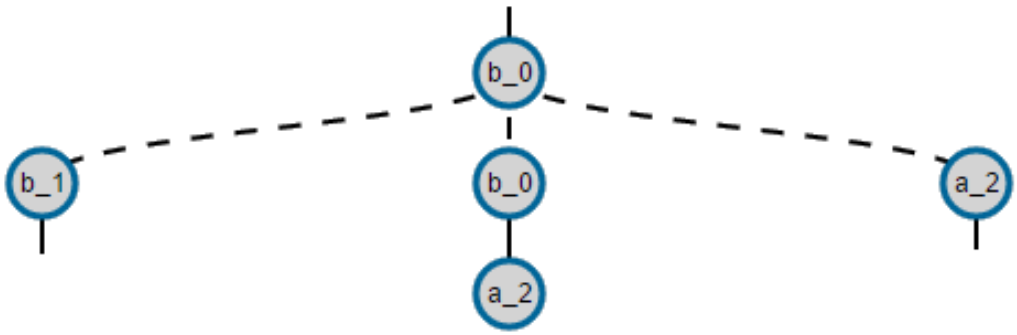


Figure 5.9: Formula-Tree of the Principal Inhabitant's Type

The sub-tree of the formula-tree that has not been used, was replaced by a leaf node with a new type-variable b_1*. The principal type of the term-scheme* $\lambda x_1 x_2 x_3 . x_2 x_3$ *is* $b\_1 \to (a\_2 \to b\_0) \to a\_2 \to b\_0$*.*
To obtain the formula-tree of the principal type of this proof-tree, it is necessary to apply some changes in the previous formula-tree. Firstly, the tool needs to choose an element of each equivalence class to name all the elements of that same class. The tool always chooses the first element of each equivalence class. Then, if a primitive part has not been used, and consequently its descendant primitive parts, the tool will replace that sub-tree with a new primitive part (a leaf node) whose node name is the same as the first node of the removed sub-tree.

In the following we will see how this feature was implemented.
The tool starts by changing the types in the nodes of the type's formula-tree. The type-variables are numbered in order to distinguish them and a different equivalence class is assigned to each node. To calculate the number of each type-variable, the tool uses the object treeJSON from Section 4.3. It is necessary to add two new fields to the object, cPart and color. The cPart contains the number of the occurrence of the type-variable of the node. The color contains the color of each node. The tool builds the formula-tree of the type, using the same process that was used in Section 4.4 but now, instead of the node field, it uses the cPart field to assign a name to each node. Everytime a new primitive part is added to the proof-tree, it is necessary to verify if the tail-variable where the primitive part is being added and the head-variable

of the primitive part are in the same equivalence class. If they are not, then their equivalence classes are merged. To be able to remove primitive parts from the proof-tree, we maintain an array with all the equivalences, i.e. substitutions, between nodes in the proof-tree. We also added a new parameter **old** to each node of the proof-tree, in order to know which node was replaced when each primitive part was added to the proof-tree. Thus, if a primitive part whose head-variable is b_1 is removed, and its **old** value is b_2, it means that we are removing an occurrence of $b\_2 = b\_1$ from the equivalence array. Then, the array that keeps the equivalences is used to update the equivalence classes.

5.1.1 Generation of Principal Inhabitants

It is well known [24] that every inhabited type α is a principal type for some λ-term. However, α might have no principal inhabitant in normal form, cf. Example 5.1.3.

Example 5.1.3. *Type $a \to (b \to b) \to a$ is the principal type of λ-term $\lambda xy.(\lambda uv.u)x(\lambda z.y(yz))$. However, it is not the principal type of any term in β-normal form.*
In fact, by inspection of its formula-tree in Figure 5.10, one can easily conclude that X_2 can not be used in any proof-tree.

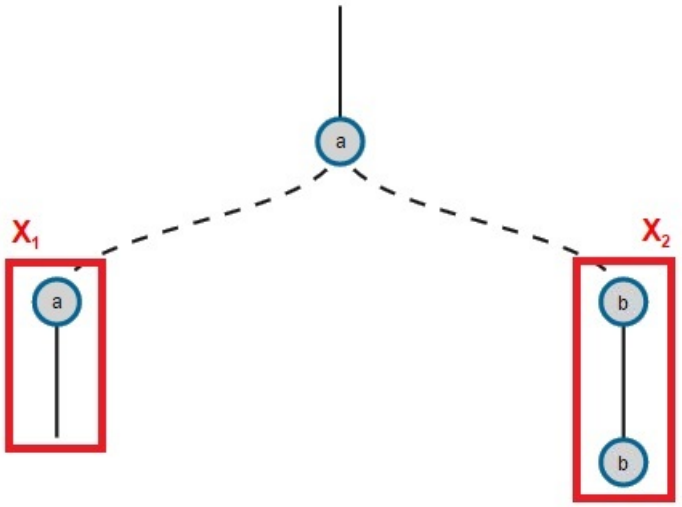


Figure 5.10: Formula-Tree

In order to address this matter and search for principal inhabitants in β-normal form automatically, one can implement an algorithm similar to the ones that we have created for the automatic generation of the minimal proof-trees, or proof-trees in the different subsystems.

Note that we want to verify if there exists a principal inhabitant in β-normal form, we are not trying to generate all principal inhabitants. For this, before adding a primitive part to the proof-tree, we verify if that primitive part has already been added in the same conditions, i.e. in the same context, with the same equivalence classes and with the same set of used primitive parts. In that case, the primitive part is discarded because we do not want to add unnecessary primitive parts to the proof-tree. Furthermore, this restriction guaranties the termination of the algorithm. In the end of each proof-tree construction, we verify if all composed primitive parts (primitive parts with tail variables) have been used, and if the equivalence classes are the expected ones. In the following we will present the algorithm, but first list the used variables and functions:

Variables

- *node* is a node of a primitive part of the formula-tree. Initially is the tail variable of the root-node of the formula-tree;
  - *node.history* is a list containing the sequence of primitive parts used in the current branch of the proof-tree. For each primitive part in the sequence, it stores the correspondent context, the equivalence classes and the set of used primitive parts. Initially it is an empty list.
  - *node.context* is the context of the *node*.
- *nodes* is a list with the remaining nodes to which the algorithm will be applied. Initially it is an empty list;
- *eqClasses* is a list that keeps the current set of equivalence classes. Initially it contains one class for each node of the formula-tree.
- *usedPParts* is a list that keeps the primitive parts that have already been used. Initially it is an empty list;
- *hasPInhab* is the algorithm's boolean return value. Initially it is false.

Functions

- `getAvailablePParts`(*node, context*) is a function that returns the primitive parts in context whose head-variable matches the node's type;
- `repeatedHistory`(*pPart, history*) is a function that verifies if the primitive part *pPart* was already used in the same conditions in *history*. It returns a boolean value;
- `updatedEqClasses`(*eqClasses, node, pPart*) is a function that merges the classes of the type of *node* and the head variable of the primitive part *pPart* in *eqClasses*. It returns the updated equivalence classes;
- `addPPart`(*usedPParts, pPart*) is a function that returns the list resulting from adding the primitive part *pPart* to the list *usedPParts*.
- `obtainContext`(*pPart, pPartNode, context*) is a function that updates the context by adding the primitive parts descending from *pPartNode*, which is a tail variable of *pPart*;
- `updateHistory`(*pPart, context, eqClasses, usedPParts, history*) is a function that returns the object resulting from adding an entry with the tuple (*pPart, context, eqClasses, usedPParts*) to *history*;
- `obtainNewNode`(*part, context, history*) is a function that returns a node resulting from adding a *context* and an *history* to a *part*.
- `completedEqClasses`(*eqClasses*) is a function that verifies if the equivalence classes are the expected ones. It returns a boolean value;
- `allNecessaryPartsUsed`(*usedParts*) is a function that verifies if all the composed primitive parts of the formula-tree have been used. It returns a boolean value;

Algorithm - Search for Principal Inhabitants

```
Function searchInhabitant(node, nodes, eqClasses, usedPParts, hasPInhab)
    pParts = getAvailablePParts(node, node.context);
    for (each pPart in pParts) do
        repeatedHistory = repeatedHistory(pPart, node.history);
        if (!repeatedHistory) then
            newEqClasses = updatedEqClasses(eqClasses, node, pPart);
            newUsedPParts = addPPart(usedPParts, pPart);
            newNodes = nodes;
            for (each tail in pPart.tail) do
                newContext = obtainContext(pPart, tail, node.context);
                newHistory = updateHistory(pPart, node.context, eqClasses,
                usedPParts, node.history);
                newNode = obtainNewNode(tail, newContext, newHistory);
                newNodes.push(newNode);
            end
            if (newNodes == empty) then
                completedEqClasses = completedEqClasses(newEqClasses);
                allNecessaryPartsUsed = allNecessaryPartsUsed(newUsedParts);
                if (allNecessaryPartsUsed and completedEqClasses) then
                    hasPInhab = true;
                end
            end
            else
                newNode = newNodes.head;
                hasPInhab = searchInhabitant(newNode, newNodes.tail,
                newEqClasses, newUsedPParts, hasPInhab);
            end
        end
        if (hasPInhab) then
            break;
        end
    end
    return hasPInhab;
end
```

5.2 Grammar

In [26] it was shown that it is possible to describe the set of normal inhabitants of a type α using an infinite extension of the concept of Context-Free Grammar (CFG), which allows for an infinite number of non-terminal symbols as well as production rules. In [10, 8] it was shown that it is possible to describe the set of normal inhabitants of α using the concept of CFG. For every α a CFG G_α is defined, which generates all possible term-schemes for α.
If the option **Type's Grammar** is chosen, which is the second option of the tool, the CFG for the given type is generated by the tool.
The initial symbol of G_α is S. The terminal symbols will correspond to the variable names $x_1, x_2, \ldots$ corresponding to primitive parts $X_1, X_2, \ldots$. The non-terminal symbols are of the form A^X, where A corresponds to an atomic type a, and X is the associated context.
In the following we will explain how the tool generates the context-free grammar of the given type. In the construction of the grammar the tool will need the JSON objects that were described in Section 4.3. We will now see an example.

Example 5.2.1. *Consider type* $\alpha = ((a \to b) \to a \to b) \to (a \to b) \to a \to b$ *and its formula-tree:*

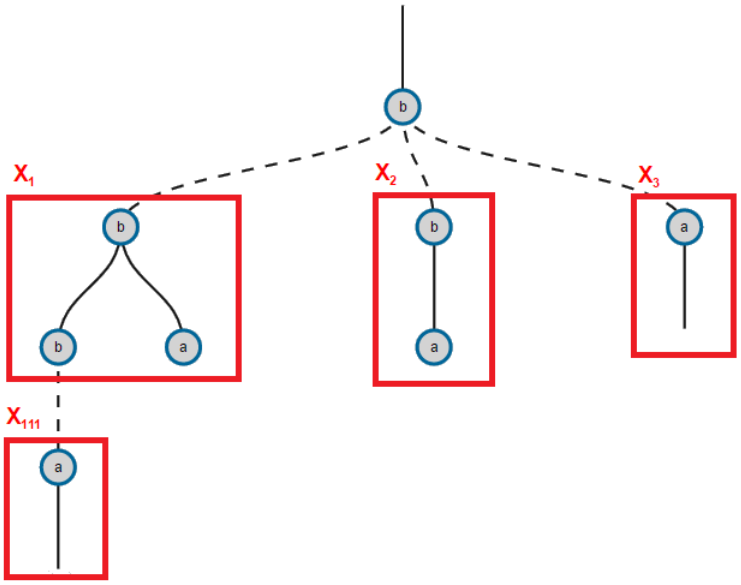


Figure 5.11: Formula-Tree

The initial rule is $\lambda x_1 x_2 x_3.\Delta$*, where* Δ *must have type* b*, and context* $\{x_1, x_2, x_3\}$*, which are the root node type, and the primitive parts that descend directly form the*

root node, respectively. Thus, the grammar's initial rule is:

$$S \rightarrow \lambda x_1 x_2 x_3 . B^{\{x_1,x_2,x_3\}}$$

The abstraction sequence $\lambda x_1 x_2 x_3.$ represents the primitive parts that are available in the very beginning of the grammar construction, i.e. the direct descendants of the head of the formula-tree. $B^{\{x_1,x_2,x_3\}}$ is the next non-terminal, which corresponds to a node of type b. Primitive parts X_1, X_2, and X_3 are available. Only two of those three primitive parts have type b on their head nodes, namely X_1 and X_2, which means that $B^{\{x_1,x_2,x_3\}}$ will have two rules. If primitive part X_1 is chosen, we can see that, in the partsJSON object, it has two children, the first child id is 1_1, which has type b, and the second child id is 1_2, which has type a. In the treeJSON object, we can see that 1_1 has a child, which is node 1_1_1, this means that will be necessary to add a new available primitive part to that non-terminal, primitive part X_{111}. However, node 1_2 has no children, so it will keep the available primitive parts from the previous non-terminal. So the following rule is obtained:

$$B^{\{x_1,x_2,x_3\}} \rightarrow x_1(\lambda x_{111} . B^{\{x_1,x_2,x_3,x_{111}\}}) A^{\{x_1,x_2,x_3\}}$$

The λx_{111} shows that a new available primitive part was added, and which non-terminals are affected by it.
If the primitive part X_2 is chosen, by repeating the previous process, we can see that the primitive part X_2 has one child, whose type is a, and this child has no children. So the following rule is obtained:

$$B^{\{x_1,x_2,x_3\}} \rightarrow x_2 A^{\{x_1,x_2,x_3\}}$$

Now, there are two new non-terminals, $B^{\{x_1,x_2,x_3,x_{111}\}}$ and $A^{\{x_1,x_2,x_3\}}$. By repeating the process as many times as necessary, the following grammar is obtained by the algorithm:

$$\begin{array}{lcl}
S & \rightarrow & \lambda x_1 x_2 x_3 . B^{\{x_1,x_2,x_3\}} \\
B^{\{x_1,x_2,x_3\}} & \rightarrow & x_1(\lambda x_{111} . B^{\{x_1,x_2,x_3,x_{111}\}}) A^{\{x_1,x_2,x_3\}} \mid x_2 A^{\{x_1,x_2,x_3\}} \\
B^{\{x_1,x_2,x_3,x_{111}\}} & \rightarrow & x_1(\lambda x_{111} . B^{\{x_1,x_2,x_3,x_{111}\}}) A^{\{x_1,x_2,x_3,x_{111}\}} \mid x_2 A^{\{x_1,x_2,x_3,x_{111}\}} \\
A^{\{x_1,x_2,x_3\}} & \rightarrow & x_3 \\
A^{\{x_1,x_2,x_3,x_{111}\}} & \rightarrow & x_3 \mid x_{111}
\end{array}$$

If the option **Type's Grammar** *for α is chosen, the tool will generate a simplified grammar.*
To compute the simplified grammar we apply some changes to the grammar generated by the algorithm. The first non-terminal symbol will remain as an S, but a different letter will be assigned to each other non-terminal symbol. Then, the rules consisting only of terminal symbols can be removed and their occurrences in the other rules are replaced by their non-terminal symbols.

The simplified grammar is:

$$\begin{array}{lcl} S & \rightarrow & \lambda x_1x_2x_3.A \\ A & \rightarrow & x_1(\lambda x_{111}.B)x_3 \mid x_2x_3 \\ B & \rightarrow & x_1(\lambda x_{111}.B)x_3 \mid x_2x_3 \mid x_1(\lambda x_{111}.B)x_{111} \mid x_2x_{111} \end{array}$$

This grammar generates the following infinite set of term-schemes of α.

$$\begin{aligned} \mathcal{L}(G_\alpha) = \{&\lambda x_1x_2x_3.x_2x_3, \\ &\lambda x_1x_2x_3.x_1(\lambda x_{111}.x_2x_3)x_3, \\ &\lambda x_1x_2x_3.x_1(\lambda x_{111}.x_2x_{111})x_3, \\ &\lambda x_1x_2x_3.x_1(\lambda x_{111}.(\lambda x_{111}.x_2x_3)x_3)x_3, \\ &\lambda x_1x_2x_3.x_1(\lambda x_{111}.(\lambda x_{111}.x_2x_{111})x_3)x_3, \\ &\lambda x_1x_2x_3.x_1(\lambda x_{111}.(\lambda x_{111}.x_2x_3)x_{111})x_3, \\ &\lambda x_1x_2x_3.x_1(\lambda x_{111}.(\lambda x_{111}.x_2x_{111})x_{111})x_3, \ldots\} \end{aligned}$$

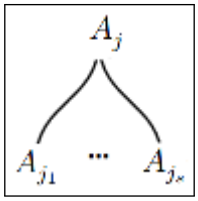
In [10] G_α was formally defined as follows.

Definition 5.2.1. Let α be a type, with atoms $A_1, \ldots, A_m$ and such that FT_α has primitive parts $x, x_{t_1}, \ldots, x_{t_n}$, where x is the root-node of FT_α. Let $G_\alpha = (T, N, R, S)$ be the context-free grammar with

- a set of terminal symbols $T = \{(,), \lambda, ., x_1, \ldots, x_n\}$;
- a set of non-terminal symbols $N = \{S\} \cup \{A_i^X \mid 1 \leq i \leq n, X \in 2^{\{x_1,\ldots,x_n\}}\}$;
- a start symbol S;
- and R is the smallest set satisfying the following conditions:
  - If $x =$  has direct descendants $x_{t_1}, \ldots, x_{t_k}$ in FT_α where

$1 \leq k \leq n$, then R has exactly one production rule for S which is $S \to \lambda x_{t_1} \ldots x_{t_k}.A_s^{\{x_{t_1},\ldots,x_{t_k}\}}$;

- whenever a non-terminal symbol A_j^X appears on the right side of a production rule in R, then for every $x_\iota \in X$ such that x_ι is of the form  there is a rule in R of the form $A_j^X \to x_\iota$;
- whenever a non-terminal symbol A_j^X appears on the right side of a production rule in R, then for every $x_\iota \in X$ such that x_ι is of the form



and such that in FT_α each A_{j_l} has direct descendants $x_{\iota_l^1}, \ldots, x_{\iota_l^{t_l}}$, there is a rule in R of the form $A_j^X \to x_\iota(\lambda x_{\iota_1^1} \ldots x_{\iota_1^{t_1}}.A_{j_1}^{X_1}) \ldots (\lambda x_{\iota_s^1} \ldots x_{\iota_s^{t_s}}.A_{j_s}^{X_s})$ where $X_l = X \cup \{\iota_l^1, \ldots, \iota_l^{t_l}\}$, for $1 \leq l \leq s$.

We will now adapt this formal definition in order to provide formal definitions for grammars in the **BCI**- and in the **BCK**-subsystem, respectively.

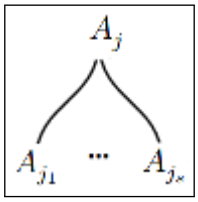
5.2.1 Grammar for the BCI- and the BCK-subsystems

There are some differences between the grammar in the complete system of λ-calculus and in the ones of the subsystems. In the **BCI**-subsystem we need to guarantee that each primitive part is used exactly once while in the **BCK**-subsystem each piece can be used at most once.
We can formally define G_α for the **BCI**-subsystem as follows:

Definition 5.2.2. Let α be a type, with atoms $A_1, \ldots, A_m$ and such that FT_α has primitive parts $x, x_{t_1}, \ldots, x_{t_n}$, where x is the root-node of FT_α. Let $G_\alpha = (T, N, R, S)$ be the context-free grammar with

- a set of terminal symbols $T = \{(,), \lambda, ., x_1, \ldots, x_n\}$;
- a set of non-terminal symbols $N = \{S\} \cup \{A_i^X \mid 1 \leq i \leq n, X \in 2^{\{x_1,\ldots,x_n\}}\}$;
- a start symbol S;

- and R is the smallest set satisfying the following conditions:
  - If $x =$  has direct descendants $x_{t_1}, \ldots, x_{t_k}$ in FT_α where $1 \leq k \leq n$, then R has exactly one production rule for S which is $S \to \lambda x_{t_1} \ldots x_{t_k}.A_s^{\{x_{t_1},\ldots,x_{t_k}\}}$;
  - whenever a non-terminal symbol A_j^X appears on the right side of a production rule in R, then for every $x_\iota \in X$ such that x_ι is of the form  there is a rule in R of the form $A_j^X \to x_\iota$, iff $X = \{x_\iota\}$;
  - whenever a non-terminal symbol A_j^X appears on the right side of a production rule in R, then for every $x_\iota \in X$ such that x_ι is of the form

    and such that in FT_α each A_{j_l} has direct descendants $x_{\iota_l^1}, \ldots, x_{\iota_l^{r_l}}$, there is a rule in R of the form $A_j^X \to x_\iota(\lambda x_{\iota_1^1} \ldots x_{\iota_1^{r_1}}.A_{j_1}^{X_1}) \ldots (\lambda x_{\iota_s^1} \ldots x_{\iota_s^{r_s}}.A_{j_s}^{X_s})$ for all possible sets $X_1, \ldots, X_s$, such that for $1 \leq k, k' \leq s$, $X_k \neq \emptyset$, $X_k = \{x_{\iota_k}^1, \ldots, x_{\iota_k}^{r_k}\} \cup X'_k$, $X'_k \cap X'_{k'} = \emptyset$, $\bigcup_{1 \leq j \leq s} X'_j = X \backslash \{x_\iota\}$.

In the following we will see an example of a **BCI**-subsystem grammar:

Example 5.2.2. *Consider the type* $\alpha = ((a \to b) \to a \to b) \to (a \to b) \to a \to b$.

The correspondent grammar is:

$$
\begin{array}{lll}
S & \to & \lambda x_1 x_2 x_3.B^{\{x_1,x_2,x_3\}} \\
B^{\{x_1,x_2,x_3\}} & \to & x_1(\lambda x_{111}.B^{\{x_3,x_{111}\}})A^{\{x_2\}} \mid x_1(\lambda x_{111}.B^{\{x_2,x_{111}\}})A^{\{x_3\}} \mid \\
 & & x_1(\lambda x_{111}.B^{\{x_{111}\}})A^{\{x_2,x_3\}} \mid x_2 A^{\{x_1,x_3\}} \\
B^{\{x_2,x_{111}\}} & \to & x_2 A^{\{x_{111}\}} \\
A^{\{x_3\}} & \to & x_3 \\
A^{\{x_{111}\}} & \to & x_{111}
\end{array}
$$

That can be simplified to:

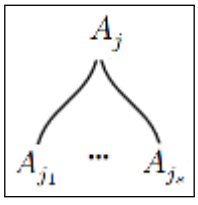
$S \quad \rightarrow \quad \lambda x_1 x_2 x_3 . x_1 (\lambda x_{111} . x_2 x_{111}) x_3$

Note that the rules that do not generate anything have disappeared.

We can formally define G_α for the **BCK**-subsystem as follows:

Definition 5.2.3. Let α be a type, with atoms $A_1, \ldots, A_m$ and such that FT_α has primitive parts $x, x_{t_1}, \ldots, x_{t_n}$, where x is the root-node of FT_α. Let $G_\alpha = (T, N, R, S)$ be the context-free grammar with

- a set of terminal symbols $T = \{(,), \lambda, ., x_1, \ldots, x_n\}$;
- a set of non-terminal symbols $N = \{S\} \cup \{A_i^X \mid 1 \leq i \leq n, X \in 2^{\{x_1, \ldots, x_n\}}\}$;
- a start symbol S;
- and R is the smallest set satisfying the following conditions:
  - If $x = \boxed{\begin{array}{c} | \\ A_s \end{array}}$ has direct descendants $x_{t_1}, \ldots, x_{t_k}$ in FT_α where $1 \leq k \leq n$, then R has exactly one production rule for S which is $S \rightarrow \lambda x_{t_1} \ldots x_{t_k} . A_s^{\{x_{t_1}, \ldots, x_{t_k}\}}$;
  - whenever a non-terminal symbol A_j^X appears on the right side of a production rule in R, then for every $x_\iota \in X$ such that x_ι is of the form $\boxed{\begin{array}{c} A_j \\ | \end{array}}$ there is a rule in R of the form $A_j^X \rightarrow x_\iota$;
  - whenever a non-terminal symbol A_j^X appears on the right side of a production rule in R, then for every $x_\iota \in X$ such that x_ι is of the form

    and such that in FT_α each A_{j_l} has direct descendants $x_{\iota_l^1}, \ldots, x_{\iota_l^{r_l}}$, there is a rule in R of the form $A_j^X \rightarrow x_\iota (\lambda x_{\iota_1^1} \ldots x_{\iota_1^{r_1}} . A_{j_1}^{X_1}) \ldots (\lambda x_{\iota_s^1} \ldots x_{\iota_s^{r_s}} . A_{j_s}^{X_s})$

for all possible sets $X_1, \ldots, X_s$, such that for $1 \leq k, k' \leq s$, $X_k \neq \emptyset$, $X_k = \{x^1_{\iota_k}, \ldots, x^{r_k}_{\iota_k}\} \cup X'_k$, $X'_k \cap X'_{k'} = \emptyset$, $\bigcup_{1 \leq j \leq s} X'_j = X \backslash \{x_\iota\}$.

In the following we will see an example of a **BCK**-subsystem grammar:

Example 5.2.3. *Consider the type* $\alpha = ((a \to b) \to a \to b) \to (a \to b) \to a \to b$.

The correspondent grammar is:

$$
\begin{array}{lll}
S & \to & \lambda x_1 x_2 x_3 . B^{\{x_1,x_2,x_3\}} \\
B^{\{x_1,x_2,x_3\}} & \to & x_1(\lambda x_{111} . B^{\{x_3,x_{111}\}}) A^{\{x_2\}} \mid x_1(\lambda x_{111} . B^{\{x_2,x_{111}\}}) A^{\{x_3\}} \mid \\
 & & x_1(\lambda x_{111} . B^{\{x_{111}\}}) A^{\{x_2,x_3\}} \mid x_2 A^{\{x_1,x_3\}} \\
B^{\{x_2,x_{111}\}} & \to & x_2 A^{\{x_{111}\}} \\
A^{\{x_3\}} & \to & x_3 \\
A^{\{x_{111}\}} & \to & x_{111} \\
A^{\{x_1,x_3\}} & \to & x_3
\end{array}
$$

That can be simplified to:

$$
\begin{array}{lll}
S & \to & \lambda x_1 x_2 x_3 . x_1(\lambda x_{111} . x_2 x_{111}) x_3 \mid \lambda x_1 x_2 x_3 . x_2 x_3
\end{array}
$$

5.3 Choose a Long

So far, we saw how to build proof-trees, obtain their respective term-scheme, their long inhabitants, and their η-families from a type. But the tool also allows us to do the reverse process.

If the option **Choose a Long** is chosen, which is the third option of the tool, the tool will calculate if a given λ-term is a long inhabitant of the type. In this case, the tool will give us the respective term-scheme, and the options to see the proof-tree, and the list of long inhabitants and their η-families.

In the following we will see how the tool works, using an example.

Example 5.3.1. *Consider type* $\alpha = ((a \to b) \to a \to b) \to (a \to b) \to a \to b$, *and the formula-tree of the previous section in Figure 5.11.*

If the following λ*-term is chosen*

$$\lambda xy.x(\lambda z.xyz),$$

then the first thing to do is to rename all the variables of the λ*-term. After renaming the tool obtains the following* λ*-term:*

$$\lambda x_1 x_2 . x_1(\lambda x_3 . x_1 x_2 x_3)$$

Now, the tool uses the JSON objects from Section 4.3 to verify if the λ-term is an inhabitant of the type, and, if so, generates the respective term-scheme.
The formula-tree will only be used to facilitate the explanation, the tool does not use the formula-tree, it uses the JSON objects.
From the JSON objects, or by looking at the formula-tree of the type, it is possible to see that the root node of the formula-tree has three direct descendants (X_1, X_2 and X_3) and the λ-term only has two, this means that the tool applies η-expansion to the λ-term. After applying η-expansion the tool obtains the following λ-term:

$$\lambda x_1 x_2 x_4 . x_1 (\lambda x_3 . x_1 x_2 x_3) x_4$$

Now, the tool breaks the λ-term in three parts:
head*:* $x_1 x_2 x_4$
primitive partVariable*:* x_1
tail*:* $\lambda x_3 . x_1 x_2 x_3; x_4$
Note that **tail** *has length 2, meaning that it has two subterms.*
The **primitive partVariable** x_1 *corresponds to the first variable in* **head**. *This means that it will correspond to the first descendant of the root node of the formula-tree whose id is* 1. *Now the tool verifies if this element has two children, which is the length, i.e. number of elements, of the* **tail**. *If it does, then the tool can proceed on the term-scheme construction. Otherwise, if we consult the* partsJSON *object, we can see that element* 1 *has two children,* 1_1 *and* 1_2. *This means that* 1_1, *which has type b, will correspond to* $\lambda x_3 . x_1 x_2 x_3$, *which is the first element of the* **tail**, *and* 1_2, *which has type a, will correspond to* x_4, *which is the second element of the* **tail**. *By looking at the* treeJSON *object, we can see that the element* 1_1 *has a child,* 1_1_1. *This means that it could be necessary to introduce a new available primitive part, and that* 1_2 *has no children, so there are no new available primitive parts to introduce. We can see that* $\lambda x_3 . x_1 x_2 x_3$ *already has a lambda with one variable, thus there is no need to introduce a new variable. And* x_4 *is a variable which does not introduce any new available primitive part. The term-scheme has not changed:*

$$\lambda x_1 x_2 x_4 . x_1 (\lambda x_3 . x_1 x_2 x_3) x_4$$

Now, since x_4 *is only a variable, we just need to analyse* $\lambda x_3 . x_1 x_2 x_3$, *which is our new term. Note that we will keep the previous head and add to it the new available primitive parts, if they exist.*
Now, the tool breaks the λ-term in three parts:
head*:* $x_1 x_2 x_4 x_3$
primitive partVariable*:* x_1

tail: $x_2; x_3$
Note that **tail** *has length 2, meaning that it has two subterms.*
The **primitive partVariable** x_1 *corresponds to the first variable in the* **head**, *it will correspond to the first descendant of the root node of the formula-tree whose id is* 1. *Now it is necessary to verify if this element has two children, which is the length of the* **tail**. *If it does, then the tool can proceed on the term-scheme construction. If we consult the* partsJSON *object, we can see that element* 1 *has two children,* 1_1 *and* 1_2. *This means that* 1_1, *which has type b, will correspond to* x_2, *and* 1_2, *which has type a, will correspond to* x_3. *If we look at the* treeJSON *object, we can see that element* 1_1 *has a child,* 1_1_1. *This means that could be necessary introduce a new available primitive part, and we can also see that* 1_2 *has no children, so there are no new available primitive parts to introduce. Now, it will be necessary to introduce a new available primitive part to* x_2, *this means that after applying* η*-expansion to* x_2, *we will obtain* $\lambda x_5.x_2x_5$. *But* x_4 *is a variable which does not introduce any new available primitive part. We now have:*
head: $x_1x_2x_4x_3$
primitive partVariable: x_1
tail: $\lambda x_5.x_2x_5; x_3$
The term-scheme has changed to:

$$\lambda x_1x_2x_4.x_1(\lambda x_3.x_1(\lambda x_5.x_2x_5)x_3)x_4$$

Now, since x_3 *is only a variable, the tool just needs to analyse* $\lambda x_5.x_3x_5$, *that is the new term. Note that the previous head will be kept, and to it will be added new available primitive parts, if they exist.*
Now, the tool breaks the λ*-term in three parts:*
head: $x_1x_2x_4x_3x_5$
primitive partVariable: x_2
tail: x_5
Note that **tail** *has length 1, meaning that it has one subterm.*
The **primitive partVariable** x_2 *corresponds to the second variable in the* **head**, *this means that it will correspond to the second descendant of the root node of the formula-tree whose id is* 2. *Now it is necessary to verify if this element has one child, which is the length of the* **tail**. *If it does, then the tool can proceed on the term-scheme construction. If we consult the* partsJSON *object, we can see that element* 2 *has one child,* 2_1. *This means that* 2_1, *which has type a, will correspond to* x_5. *If we look at the* treeJSON *object, we can see that* 2_1 *has no children, so there are no new available primitive parts to introduce. Thus,* x_5 *is a variable which does not introduce any new*

available primitive part. The term-scheme has not changed:

$$\lambda x_1 x_2 x_4 . x_1 (\lambda x_3 . x_1 (\lambda x_5 . x_2 x_5) x_3) x_4$$

Now, since x_5 is only a variable, i.e. does not depend of a λ, there is no need to analyse it.
Now that the tool has generated the term-scheme. After renaming it according to the identifiers of JSON objects, the tool obtains:

$$\lambda x_1 x_2 x_3 . x_1 (\lambda x_{111} . x_1 (\lambda x_{111} . x_2 x_{111}) x_{111}) x_3$$

Finally, it is necessary to verify if all the types are the correct ones, by doing an analysis similar to the previous one.
Now it is possible to choose between visualizing the proof-tree correspondent to the obtained term-scheme, or its long inhabitants and their η-families.

Chapter 6

Conclusions and Future Work

This thesis describes the design and implementation of an interactive tool that explores the potentialities of the Formula-Tree Method, in order to provide assistance for the study of type-inhabitation, or equivalently provability of formulas in the implicational fragment of intuitionistic propositional logic.
The tool implements the core features of the Formula-Tree Method, such as the construction of the formula-tree of a type, the construction of proof-trees, and the computation of long inhabitants of the type. We implemented a feature that constructs a context-free grammar for a given type, from which all of the type's term schemes can be generated, and as a result all of its long inhabitants and η-families can be obtained. Additionally we implemented a minimal proof-trees feature that automatically builds the minimal proof-trees of a type, and calculates the number of its normal inhabitants. The tool also allows the construction and automatic generation, of proof-trees for three λ-calculus subsystems, the **BCIW**-, the **BCI**- and the **BCK**-subsystems. The tool allows the user to focus on principal normal inhabitants. Finally, it is possible to verify if a given λ-term is an inhabitant of a type and if so, display the respective term-schemes, proof-tree and long inhabitants and their η-families.
With this work we were able to improve a previous tool in terms of portability, usability and technology. Additionally we added some other functionalities. Nowadays, there are more advanced and compositional technologies to address this kind of problem than the ones used in the former application. Therefore, it made sense to implement the new application from scratch. The technologies we used were HTML, CSS, jQuery, PHP, Bootstrap and D3.js.
As future work, it would be interesting to implement the features we defined in Sections 5.2.1 and 5.1.1. The former consists of an algorithm to construct context-free grammars for the **BCI**- and the **BCK**-subsystems. The number of inhabitants of a type in these subsystems are always finite. Thus, after simplification, the obtained

grammars will be of the form $S \to M_1 \mid \ldots \mid M_n$, where $\{M_1, \ldots, M_n\}$ is exactly the set of long normal inhabitants of α. Although these algorithms were not implemented, it is possible to obtain the set of long normal inhabitants of α using the option **Generate Possible Proof-Trees** when we are constructing proof-trees for the **BCI**- and the **BCK**-subsystems. The algorithm in Section 5.1.1 enables the user to decide if a given type has a principal inhabitant in β-normal form. Finally, it would be interesting to implement an algorithm for the generation of the normal inhabitants of a type α from its grammar G_α.

References

[1] Formula tree lab. `http://www.dcc.fc.up.pt/~sbb/FTLab/ftlab/`. Accessed: 2015-09.

[2] Formula-tree method tool. `http://www.alunos.dcc.fc.up.pt/~up200905000/FTM/index.html`. Accessed: 2016-12.

[3] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2013.

[4] C.-B. Ben-Yelles. Type-assignment in the lambda-calculus; syntax and semantics. *PhD thesis, Mathematics Dept., University of Wales Swansea, UK.*, 1979.

[5] Bootstrap. `http://getbootstrap.com/`. Accessed: 2015-10.

[6] Pierre Bourreau and Sylvain Salvati. Game semantics and uniqueness of type inhabitance in the simply-typed λ-calculus. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, pages 61–75, 2011.

[7] Sabine Broda and Luís Damas. On the structure of normal λ-terms having a certain type. *Proc. 7th WoLLIC'2000*, pages 33–43, 2000.

[8] Sabine Broda and Luís Damas. A context-free grammar representation for normal inhabitants of types in $\mathrm{ta}_{\mathrm{lambda}}$. In Pavel Brazdil and Alípio Jorge, editors, *Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving, 10th Portuguese Conference on Artificial Intelligence, EPIA 2001, Porto, Portugal, December 17-20, 2001, Proceedings*, volume 2258 of *Lecture Notes in Computer Science*, pages 321–334. Springer, 2001.

[9] Sabine Broda and Luís Damas. Studying provability in implicational intuitionistic logic the formula tree approach. *Electr. Notes Theor. Comput. Sci.*, 67:131–147, 2002.

[10] Sabine Broda and Luís Damas. On long normal inhabitants of a type. *J. Log. Comput.*, 15(3):353–390, 2005.

[11] Sabine Broda and Luís Damas. On long normal inhabitants of a type. *J. Log. Comput.*, 15(3):353–390, 2005.

[12] Sabine Broda, Luís Damas, Marcelo Finger, and Paulo Silva e Silva. The decidability of a fragment of bb'iw-logic. *Theor. Comput. Sci.*, 318(3):373–408, 2004.

[13] Martin W. Bunder. Proof finding algorithms for implicational logics. *Theor. Comput. Sci.*, 232(1-2):165–186, 2000.

[14] Martin W. Bunder. The inhabitation problem for intersection types. In *Theory of Computing 2008. Proc. Fourteenth Computing: The Australasian Theory Symposium (CATS 2008), Wollongong, NSW, Australia, January 22-25, 2008. Proceedings*, pages 7–14, 2008.

[15] Martin W. Bunder and R. M. Rizkalla. Proof-finding algorithms for classical and subclassical propositional logics. *Notre Dame Journal of Formal Logic*, 50(3):261–273, 2009.

[16] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, pages 346–366, 1932.

[17] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[18] H. B. Curry. Functionality in combinatory logic. *Proc. Nat. Acad. Science USA*, pages 584–590, 1934.

[19] D3. `http://d3js.org/`. Accessed: 2015-10.

[20] d3noob. `http://www.d3noob.org/2014/01/tree-diagrams-in-d3js_11.html`. Accessed: 2015-10.

[21] Cluster Dendrogram. `http://bl.ocks.org/mbostock/4063570`. Accessed: 2015-10.

[22] Graphviz. `http://www.graphviz.org/`. Accessed: 2015-10.

[23] J. R. Hindley. *Basic Simple Type Theory.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press., 1997.

[24] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, December 1969.

[25] W. A. Howard. *The formulae-as-types notion of construction. In To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* J. P. Seldin and J. R. Hindley, 2012.

[26] Masako Takahashi, Yohji Akama, and Sachio Hirokawa. Normal proofs and their grammar. *Inf. Comput.*, 125(2):144–153, 1996.